

Apollo++: Automated Assessment of Learning Outcomes in Programming Projects

Master Thesis
University of Twente

Arthur Rump

Thursday, 30 November 2023

Contents

1	Introduction	3
I	Background and requirements	6
2	Background	8
2.1	Assessment starts with ILOs	8
2.2	Assessment criteria are sometimes ill-defined	9
2.3	Automated assessment tools are built for small, well-defined exercises	9
3	Stakeholders, concerns and requirements	11
3.1	Scope and environment	11
3.2	Stakeholders	12
3.3	Stakeholder interviews	13
3.3.1	Participants	15
3.3.2	Procedure	15
3.3.3	Analysis	16
3.4	Use cases	16
3.5	Requirements	22
3.6	Mission	28
II	Architecture	30
4	Components	32
5	Datamodel	36
6	Assessment pipeline	39
6.1	Type graph	42
6.2	Pattern rules	42
7	Assessment configuration	47
8	User interface	50

III	Prototype and evaluation	56
9	Prototype	58
9.1	Extractors	58
9.2	Graph matcher	59
9.2.1	Subgraph matching algorithm	61
9.3	LLM suggestions	63
10	Evaluation	65
10.1	Methods	65
10.2	Addressing requirements	67
10.3	Prototype experiments	68
10.3.1	Feasibility of the design	68
10.3.2	Appropriateness of the assessment method	69
11	Discussion	76
11.1	What works well	76
11.2	Patterns need more expressiveness	76
11.3	The pipeline could be more flexible	77
11.4	Configuration needs better support	78
IV	Conclusion and backmatter	80
12	Conclusion	81
	Acronyms	83
	Glossary	84
	References	86
	Appendices	91
A	Criteria	91
A.1	Algorithms in Creative Technology	91
A.2	Software Systems	102
B	Analysis results	119

Chapter 1

Introduction

Learning to program requires feedback, especially when it comes to complex issues like style and structure. At the University of Twente, courses tend to include a project which allows students to learn in practice and be assessed on their practical skills, but this poses a challenge for feedback: since students are working on their own projects with some freedom to make decisions, every project is different (Mader et al., 2020). In some cases this is limited to different programs that solve the same problem, while in others the functionality of the program could also take different directions between students. With a large number of students, the only way to scale this feedback practice is with a large number of tutors, but this increases the chances of inconsistent assessments (Bloxham et al., 2015; Jonsson & Svingby, 2007). Another option is to partially automate assessment, which is what we will examine in this study.

As a context, we particularly consider the project assignments given to first-year bachelor students of the Technical Computer Science (TCS) and Creative Technology (CreaTe) programmes as part of a programming course that introduces object-oriented programming. In TCS, students are tasked with implementing a multiplayer board game with a client/server architecture. Beyond what was covered in class and the rules of the game, students receive no further guidance or boilerplate code to start from. In the CreaTe course, students are free to choose what they build for the project, as long as they include topics covered in the course. Code quality plays a major part in the assessment of both projects.

Existing tools for automated assessment and intelligent tutoring systems (ITSs) aim to automate (part of) the assessment and feedback process. Many tools use automated testing techniques and only work on smaller exercises, for which there are a known number of typical solution strategies (Le & Pinkwart, 2014). These tools are difficult to adapt to larger projects, because they are configured with model solutions and/or test data, if they are configurable at all (Keuning et al., 2019). When students build projects with more than 20 classes, creating model solutions that cover all possible solutions is not a feasible endeavour.

We answer the question *How, and to what extent, can an automated assessment tool assess larger programming projects?* With larger projects we mean those projects where students are free to make significant decisions in their design of

a solution, like in the examples given before. Projects where students solve a problem from scratch and where a solution with 10 to 20 classes is reasonably sized. Projects that serve as the capstone of a course, assessing all intended learning outcomes (ILOs) covered by that course. Assessing these projects is a stepping stone towards automated assessment of ILOs throughout the course.

To answer this question, there are a few other questions to consider first, like (1) *What does assessment of projects look like, both manual and automated?* and (2) *What are the requirements for an automated assessment tool that assesses projects?* We answer these questions in part I, with the first question in chapter 2 through a review of relevant literature and the second question in chapter 3 through a requirements elicitation process with stakeholder interviews. We identified the relevant stakeholders, held interviews with some of our stakeholders to discover their concerns and from those concerns distilled the use cases and related requirements.

The main question asks how such a tool could work, so based on the requirements, we ask (3) *How to build this tool?* As an attempt to answer this question, we introduce *Apollo++*, a tool for automated assessment built specifically for larger projects. We describe the architecture for this tool in part II, with chapters 4 to 8 covering different views on the architecture.

The key features of *Apollo++* are its basis in ILOs and assessment criteria, its support for different program representations and the goal of supporting rather than replacing manual assessment. First, rather than starting from model solutions or test cases, the configuration of *Apollo++* starts with the ILOs of a course and the criteria that follow from those. Using these criteria helps to identify smaller parts of a project that are relevant to assess. Even though the assessment of a project as a whole is a complex endeavour, some criteria allow little variation and are thus easy to assess automatically (Rump & Zaytsev, 2022). Starting with ILOs builds on the approach of *Apollo* (Rump et al., 2021), which linked the ILOs of a course with abstract syntax tree (AST) patterns to provide an indication of mastery. In *Apollo++* we introduce criteria in between, to more closely reflect common assessment practice.

Second, to support a variety of criteria, *Apollo++* supports the use of different program representations. Criteria about the design of a class-hierarchy requires a different view of the program than criteria about the use of arrays and loops. These different levels of abstraction can be covered by different program representations, while using the same unified pipeline and configuration.

Finally, the goal of *Apollo++* is not to fully replace manual assessment or feedback. It will not calculate grades or make conclusions about passing or failing. The goal of the tool is to present information about the project in relation to the assessment criteria, give an indication of mastery of different ILOs and support assessment decisions made by human tutors.

As this architecture attempts to answer the question of how this tool could work, we evaluate its success in part III, answering the question: (4) *Is the design feasible and appropriate?* In other words: could a tool actually work like this in practice? To answer this question, we built a prototype of the core parts of our design, described in chapter 9. We included the core assessment pipeline and some configuration functionality, to provide a minimum viable

prototype for experiments. In chapter 10, we then evaluate the feasibility and appropriateness of the design, based on how it addresses the requirements and through experiments with the prototype on actual student projects. We discuss the results of the evaluation in chapter 11, highlighting the strengths and weaknesses of our design regarding feasibility and appropriateness, while also making suggestions for improvements.

In the closing chapter 12 we review the results from our subquestions and formulate an answer to our main question: how, and to what extent, can an automated assessment tool assess larger programming projects?

Part I

Background and requirements

This first part continues the introduction by providing background information on assessment practices, the state of the art of automated assessment and the requirements we determined for our tool. The two chapters in this part answer our first two questions: *What does assessment of projects look like, both manual and automated?* and *What are the requirements for an automated assessment tool that assesses projects?* The answers to these questions are used to guide the design for our tool, which will be introduced in part II.

Chapter 2

Background

In this chapter, we answer the question what assessment of projects looks like, considering both the manual practice and the state of the art in automated assessment tools. We will introduce intended learning outcomes and discuss the ill-definedness of assessment criteria, which are important ideas that inform the design of Apollo++. We also discuss previous work in automated assessment tools, with a focus on how they support larger projects and how they can be configured.

2.1 Assessment starts with ILOs

Intended learning outcomes (ILOs) are statements of what a student is expected to have learned at the end of a lecture, a course or study unit or even at the end of a full programme. These statements focus on student behaviour: they describe an action the student should be able to take after finishing the particular unit of study, like “*At the end of this lecture, students can ...*” or “*After following this course, you will be able to ...*” In English, the next parts of these statements will naturally be a verb and some object. The verb is used to describe the action a student should be able to take, and the object describes the relevant course topic. ILOs cover two aspects of learning: they describe the content that should be learned (the object in the sentence), as well as the level of understanding that is desired (indicated through the verb). The level of understanding is often categorised through a taxonomy like Bloom’s taxonomy (Krathwohl, 2002) or the SOLO taxonomy (Biggs & Collis, 1982).

Including a verb in an ILO not only indicates the level of understanding students are expected to acquire of that content, but it also helps to design learning activities and assessment tasks that actually teach students how to perform that activity. If our intention is for students to learn how to develop software of average size (10-20 classes), a written exam is not well-suited to assess that ability. In this case, students should actually spend time developing software during the course. If the only learning activities are lectures on the principles of object-oriented programming, it would not be surprising if students do not meet the intended outcome of being able to develop software. This principle is

called *constructive alignment*: learning activities and assessment tasks should be aligned with the ILOs of a course (Biggs & Tang, 2011).

2.2 Assessment criteria are sometimes ill-defined

Projects are different from the typical assignments assessed by automated tools, because there is no single definite solution to compare with. Open problems that do not have a definite solution are often called *ill-defined* – this indefinite endpoint is one of three criteria Simon (1978; as cited in Fournier-Viger et al., 2010) gives for calling a problem ill-defined, for example. Fournier-Viger et al. (2010) argue that ill-definedness is a continuum, ranging from well-defined to ill-defined, rather than a boolean proposition. Le et al. (2013) describe the space of ill-definedness along two axes: the number of alternative solution strategies and implementation *variability* within a strategy, and the objective solution *verifiability*.

In (Rump & Zaytsev, 2022), we extended the verifiability axis and used the model to analyse two programming projects. Rather than trying to classify a project as well- or ill-defined as a whole, we considered the assessment criteria individually. We found that even though students get a lot of freedom to make decisions in these projects, many criteria are on the well-defined side of the spectrum, making automated assessment of these criteria feasible.

While ill-defined problems are more difficult to assess, a constructivist view of education argues that these problems are necessary for learning. An open question or problem is the starting point for learning in forms of learning with a base in this tradition, like the *driving question* in project-based learning (Blumenfeld et al., 1991; Krajcik & Shin, 2014) or *wicked problems* in challenge-based learning (Gallagher & Savage, 2020; Lönngren, 2017).

2.3 Automated assessment tools are built for small, well-defined exercises

Automated assessment tools for programming assignments have existed for a long time and use a variety of techniques, but most are designed for small, well-defined exercises that can be evaluated with tests or by comparing to model solutions provided by teachers (Ala-Mutka, 2005; Ihantola et al., 2010; Le & Pinkwart, 2014; Messer et al., 2023a). Configuring these tools to support new exercises is often difficult or even impossible (Keuning et al., 2019).

More recently, tools have started using data-driven techniques (McBroom et al., 2022; Messer et al., 2023b; Paiva et al., 2022), which seems a more promising approach for ill-defined assignments. By analysing patterns in historical student submissions, these tools can learn to recognize valid solutions without requiring teachers to enumerate all possibilities. By leveraging the natural variation present in student submissions, tools can cover variation in approaches without extra configuration, as would be required for most other techniques. The only exception is automated testing, which does not care about implementation variation at

all, but that approach is limited to only objectively assessable assignments with strictly defined interfaces.

There are several approaches to using student data: clustering techniques, for example, identify groups of similar solutions (Choudhury et al., 2016; Glassman et al., 2015; Gross et al., 2012; Moghadam et al., 2015; Zhang et al., 2023), allowing teachers to efficiently provide feedback at the cluster level. Other tools extract common code patterns across submissions to highlight patterns that might prove of interest (Mens et al., 2021; Nguyen et al., 2014) or use these patterns as features to train machine learning models to identify positive and negative examples (Lazar et al., 2017; Možina & Lazar, 2018). Some data-driven tools interactively involve teachers in the learning phase, for example to mark which patterns indicate correct and incorrect solutions (Možina et al., 2018), to guide the system in finding equivalent approaches (Xu & Chee, 2003) or to define rules based on model solutions (Mitrovic, 2011; Suraweera et al., 2005).

Existing tools that work on assignments that are not objectively verifiable typically depend on heuristic techniques that only consider some aspects, rather than the full solution (Le et al., 2013). Nye et al. (2016) recommends that ill-defined domains are split into well- and ill-defined parts, such that the well-defined parts can be handled by a tool.

Recent advances in generative AI lead many to reconsider how we will teach programming in the future and also how we will assess students' programming skills (Becker et al., 2023). Experiments with GPT-3.5 show, however, that current models still struggle to provide accurate feedback on programming assignments, even on relatively small exercises (Balse et al., 2023). Even when these models improve and are able to give more accurate feedback, the lack of transparency in their workings will remain an issue when using them directly in the assessment process.

Chapter 3

Stakeholders, concerns and requirements

In this chapter, we answer our second question: what are the requirements for an automated assessment tool that assesses projects? We determined these requirements through interviews with our stakeholders, but the next section first introduces the scope and environment for the project. Then we discuss the stakeholders who have an interest in our tool (in section 3.2), how we interviewed those stakeholders to find their concerns (in section 3.3) and the use cases and requirements condensed from those concerns (in sections 3.4 and 3.5). Finally, we summarize this into a short mission statement in section 3.6.

3.1 Scope and environment

Apollo++ is designed with two programming courses at the University of Twente in mind: the programming part of the module Software Systems (SS) in the Technical Computer Science (TCS) programme and the course Algorithms in Creative Technology (AiC), which is part of the fourth module in the Creative Technology (CreaTe) programme. Both courses introduce students to object-oriented programming and let them create a final project to demonstrate their skills. In these projects, students have considerable freedom to make their own choices.

In SS, students are tasked with implementing a multiplayer board game in a client/server architecture. The rules of the game are given, but the students receive no boilerplate code or guidance beyond the generic principles covered in the course. On average, these projects contain about 24 classes of Java code (including tests). Besides the correct implementation of the functionality (following the rules of the game), the project is also assessed on the design and quality of the code, (automated) tests, documentation, and the explanations and reflection in the report.

In AiC, students learn about object-oriented structure in Processing, along with algorithms to represent physical phenomena in their animations, like flocking

and mass-spring-damper systems. Students are free to choose the topic for their project, as long as they combine at least three elements covered in the course. The final projects contain about 12 classes on average. The students are assessed primarily on complexity of the program (is there enough interaction with the user, are there at least two (non-trivial) classes, etc.), programming style and code quality. The assessment takes place during an oral examination, where students are asked to explain their project.

To scale the assessment tasks, both courses use a large group of teaching assistants, who provide feedback to students during tutorial sessions and also grade the final submissions. In AiC, a system for code sharing, Atelier, is used during these tutorials: students can upload their code when asking a question and tutors can browse the code on their own device with additional comments provided by tools integrated in the system (Fehnker et al., 2021). Both courses also use an learning management system (LMS) where students hand in assignments and the final project. Grading also happens through this LMS.

We intend Apollo++ to be used to give feedback during tutorials and for grading a submitted project. The primary users are all teaching staff. Especially for this first version, we believe that it is best for feedback to be curated by a human tutor, because they are able to provide context to students and determine what is the most relevant at a certain point. Thus, we assume that students will not interact directly with the tool, but they may receive feedback from tutors that originated in the tool.

While we will focus on this environment for our design, we intend the resulting architecture to be usable more generally, for example by treating integration with Canvas and Atelier as a more general integration problem. Still, the intended environment remains a programming course where students have relatively large freedom in the code they write, like in the mentioned projects. If a course does not contain coding assignments with such a degree of freedom, then Apollo++ is likely not the right tool and one of the simpler approaches based on model solutions or test cases might be better suited.

3.2 Stakeholders

To find the requirements for our tool, we first need to consider which stakeholders have an interest to take into account in those requirements. We identified a starter list of stakeholders and during the interviews (see section 3.3) asked the participants who else we should talk to. This resulted in the following list of stakeholders:

- **Teaching staff**, both teachers and teaching assistants. They have different relations to the tool depending on their role in a course: *coordinators* may choose to use the tool in a course, *configurators* set the tool up with the appropriate assessment configuration, *tutors “in detail”* give feedback to students on an individual level or grade a single submission, and *tutors “in overview”* want to incorporate generic feedback in plenary teaching.
- **Students** are of course stakeholders, even though they do not interact directly with the tool.

- The **examination board** is responsible for the quality of assessment and grading.
- **Program management** is responsible for the study program as a whole and the quality of content.
- **Educational support** helps educators in the organisation of their teaching, including choosing the tools they may want to use.
- **Developers and maintainers** are, of course, responsible for building the tool and adapting it to a changing environment.
- **System administrators** are responsible for deploying the tool and keeping it running and up-to-date.

These stakeholders can be organized in a power-interest matrix, to determine how we should treat the concerns of each group. In the power-interest matrix in figure 3.1, we take *power* to be the ability to make decisions whether and how the tool is used in the educational context and *interest* to be a measure of how usage directly influences a stakeholder’s (role in) education. We came to this arrangement by comparing stakeholders on the different axes. Developers/maintainers and coordinators have a similar high level of power, as they have similar power over whether and how the tool is used; teaching staff that directly uses the tool has a slightly higher interest than the coordinator, and much larger interest than anyone not using the tool directly; system administrators and educational support have similar levels of power in an organizational manner, but educational support has a slightly higher interest when it comes to educational aspects etc.

The matrix clearly shows that the teaching staff are the most important stakeholders for our project, with both high power and high interest, so their concerns should be prioritized. The examination board, system administrators, educational support and program management have lower power and interest, though they are certainly not in the bottom left corner. While teaching staff is likely to win when concerns conflict, the concerns of this group should be taken into account. To a lesser degree, this also holds for students. Opposite to how the matrix is typically read, we think the student’s concerns should be defended in the tool, since they have little power over how it is used. With conflicting concerns, however, they are unlikely to win. Lastly, developers and maintainers should be kept satisfied, but their concerns are not the most important.

3.3 Stakeholder interviews

To determine the stakeholder’s concerns, we held interviews with representatives from our stakeholders.¹ The stakeholders are or were all involved with teaching or organisation at the University of Twente, specifically the two courses described at the start of this chapter. The participants were active as teaching staff, teaching assistants, members of the examination committee or program management for one of these courses. Participants were asked about their use case for Apollo++, specifically about their concerns and the priority of those concerns. The answers were written in a document during the interview, which participants were asked to review. These interviews were not recorded.

¹These interviews were conducted according to the outlined procedure with approval from the Ethics Committee Computer & Information Science of the University of Twente. This is known to them as request *RP 2022-178*.

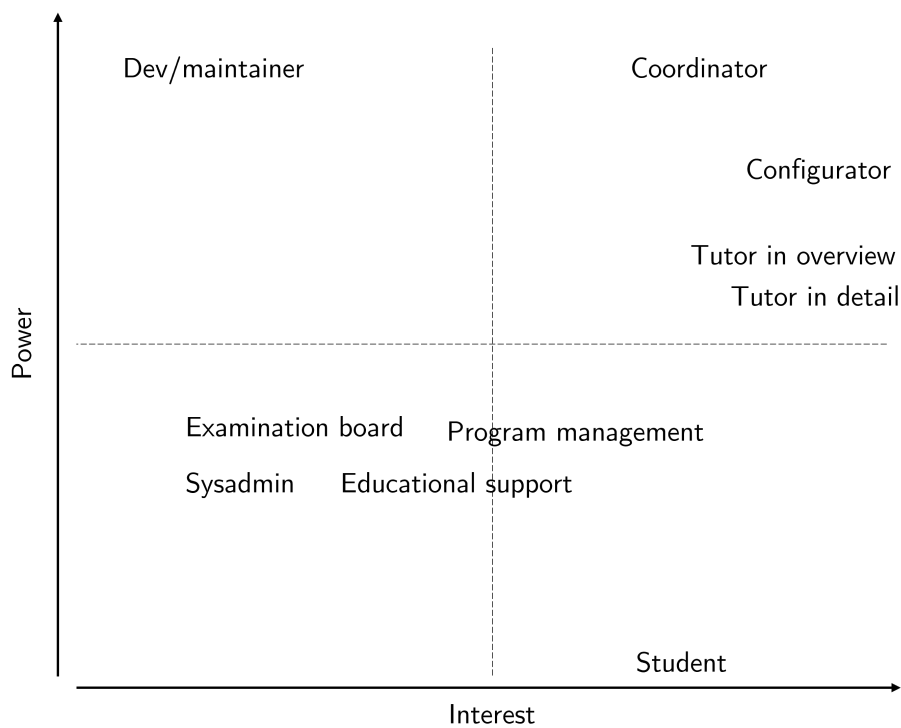


Figure 3.1: Stakeholders organised in a power-interest matrix. Stakeholders higher on the y-axis have more power and stakeholders more to the right on the x-axis have more interest.

3.3.1 Participants

We asked 19 representatives from our stakeholders to participate in an interview. Many of our participants represent multiple stakeholders, for example teaching staff who will configure the tool and use it both in detail and in overview. Participants were selected based on their publicly known role in a programming course or through personal contact and approached via a personally addressed email. 12 participated in an interview, distributed over the different groups of stakeholders as shown in figure 3.2.

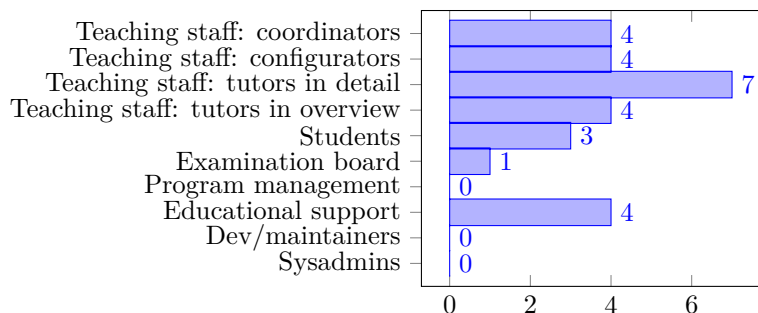


Figure 3.2: Number of participants in the interviews per type of stakeholder. Note that many participants represent more than one type of stakeholder.

Unfortunately, not all stakeholders were represented in the interviews. We based the concerns of these stakeholders on their role description, general concerns and how those could relate to the project. The *program management's* concerns were discussed with one of the supervisors of this project, who also happens to be a program director. The concerns of *developers/maintainers* were based on previous experience creating and managing *Apollo* (Rump et al., 2021). Because we only build a partial prototype in this study, we decided to not investigate the concerns of *system administrators* further, as those would likely be at the edge of our system and not important in the scope of the prototype.

3.3.2 Procedure

Each interview took about 20 to 30 minutes, depending on the number of roles the participant represented, and was conducted online or in person on campus. Each interview started with an introduction, briefing and consent. After this introduction, we asked questions about what the participant would like a tool like *Apollo++* to do or not do from their different perspectives (e.g. as a teaching assistant or as a student). Their concerns were summarised and noted down during the interview. After discussing the concerns from each perspective, the participants were asked to review the concerns that were recorded in the document and make any corrections or clarifications. Finally, we asked them to rate (1) how satisfied they would feel if the tool met that concern and (2) how dissatisfied or disappointed they would feel if the tool did not meet that concern. The interview concluded with a debriefing about the next steps and how their responses were to be used.

By taking notes and repeating back what was written, we guided our participants

into formulating concerns. This aids the validity of this method, because it ensures that we actually get the concerns from our stakeholders, rather than a story from which concerns are distilled in the analysis phase where the stakeholder’s point could more easily be misinterpreted.

We believe this format also helped to ensure the reliability of our results, because our participants were able to review the final artefact and make corrections if there were any misunderstandings during the interview. By doing the first step of analysis with the participant present, we can ensure that there is no loss of meaning or confusion in this step. We also tried to create similar starting circumstances for each participant by using the same questions, only providing suggestions from a predetermined list and not referencing concerns mentioned in previous interviews.

During the interviews we found that the satisfied/dissatisfied scoring system was difficult for our participants, especially when it comes to dissatisfied scores: some participants seemed to interpret high scores as high dissatisfaction while others interpreted low scores as being highly dissatisfied. Clarification during the interviews did not help and only led to internally inconsistent results. This means that these scores were not usable to make any conclusions and had to be discarded. Also, some participants first decided on a MoSCoW category for each concern and then tried to translate that into satisfied/dissatisfied scores, so using that system might have been more intuitive for our participants.

Another problem that arose is that some participants had difficulty separating their different roles, despite repeated guidance in the formulation of our questions. This means that a concern like “the tool supports Python and GitHub” was recorded as a concern for a *tutor in detail* even though it is not their decision to use the tool in a course or not. It might have been easier for our participants to describe their role as a *teaching staff* and distinguish in different use cases.

3.3.3 Analysis

The results of these interviews were lists of concerns for each stakeholder. To determine the requirements for our tool, we combined all similar concerns for one or multiple stakeholders. We also recorded the number of participants that voiced a certain concern to serve as a weak indicator of importance. Next, we distinguished functional requirements and quality requirements, resulting in the list of requirements can be found in table 3.1. From requirements that specify concrete actions we extracted use cases and those concrete actions, which are shown in figures 3.3 to 3.7.

In the next sections, we will describe those requirements and use cases. The next section starts with the latter, because the overview of use cases provides more structure to then understand the requirements.

3.4 Use cases

The list of use cases started from the main use cases mentioned in the scope of this project: provide individual feedback during tutorials, provide common feedback in plenary sessions and grade projects. Additionally, there is a need to

configure the tool, which we also considered as a use case. Of course, students have a use case to receive their feedback or grade.

From this starting point, we tried to map each requirement to one of those use cases. This resulted in additional use cases for teaching staff: as specialized versions of grading a project, we have “grading a project during an oral exam” and “grading a project in a grading session”, which have slightly different requirements. Additionally, teaching staff indicated they would like to help groups of students with similar needs, like those who make similar mistakes, in addition to individuals and the whole cohort. Program management and educational support, as well as teaching staff, would like to use the tool to evaluate the course based on results as well as on the constructive alignment between ILOs and assessment criteria. Based on the requirements, we also added use cases for developers/maintainers to add support for additional languages and to add support for additional types of criteria.

These use cases are shown in the use case diagram in figure 3.3. All use cases include an identifier, written between square brackets in this diagram. These identifiers will be used to refer back to the use cases, indicated by the marker *UC* where needed.

Note that not all use cases fall within the system boundary: the developer use cases do not happen within the system, but rather in development of the system. Similarly, we keep the student use cases to receive feedback and grades out of the system, because these should primarily be served by the platforms our tool integrates with. Use case *[C] Configure the tool* is not really a use case in the sense that users want to use the tool purely with the goal to configure it. We included this as a use case within the system, because stakeholders indicated that the tool should support them in the configuration phase and many actions were related to this.

Also note that the actors in these figures do not always correspond directly to the stakeholder who voiced a concern. External tools were, for example, not in the group of stakeholders we interviewed, but they are an actor in the requirement that the tool “integrates with existing platforms” voiced by coordinators. Likewise, the examination board has concerns about the tool, but they are not actors interacting with the system themselves.

Because many requirements were focused on actions our stakeholders wanted to perform, we extended the use case diagram with concrete actions. Figure 3.4 shows the actions related to configuration of the tool, figure 3.5 shows the actions related to grading and giving individual feedback, figure 3.6 shows the actions related to plenary feedback, feedback for groups and course evaluation, and figure 3.7 shows the actions related to the student use cases. In each diagram, dotted lines indicate which use cases relate to which actions. Like the use cases, actions include an identifier between square brackets, to refer to them in later chapters.

Finally, note that even though the *RF* and *RG* use cases mainly fall outside the system boundary, two detailed actions related to receiving feedback are within the system boundary, because not all external systems include the affordances needed to perform these actions. Some student-facing interfaces within the

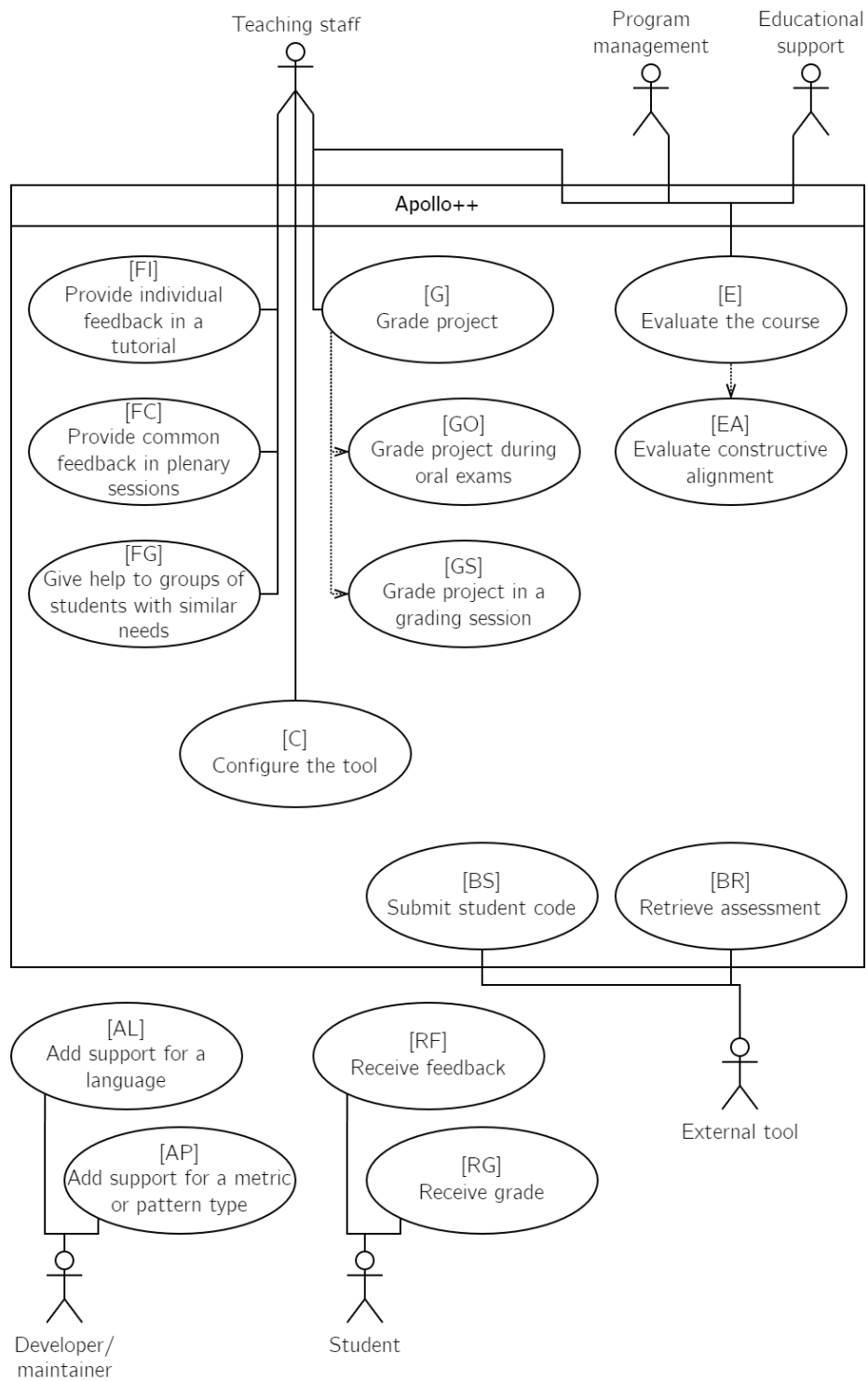


Figure 3.3: Use case diagram with the main use cases for Apollo++.

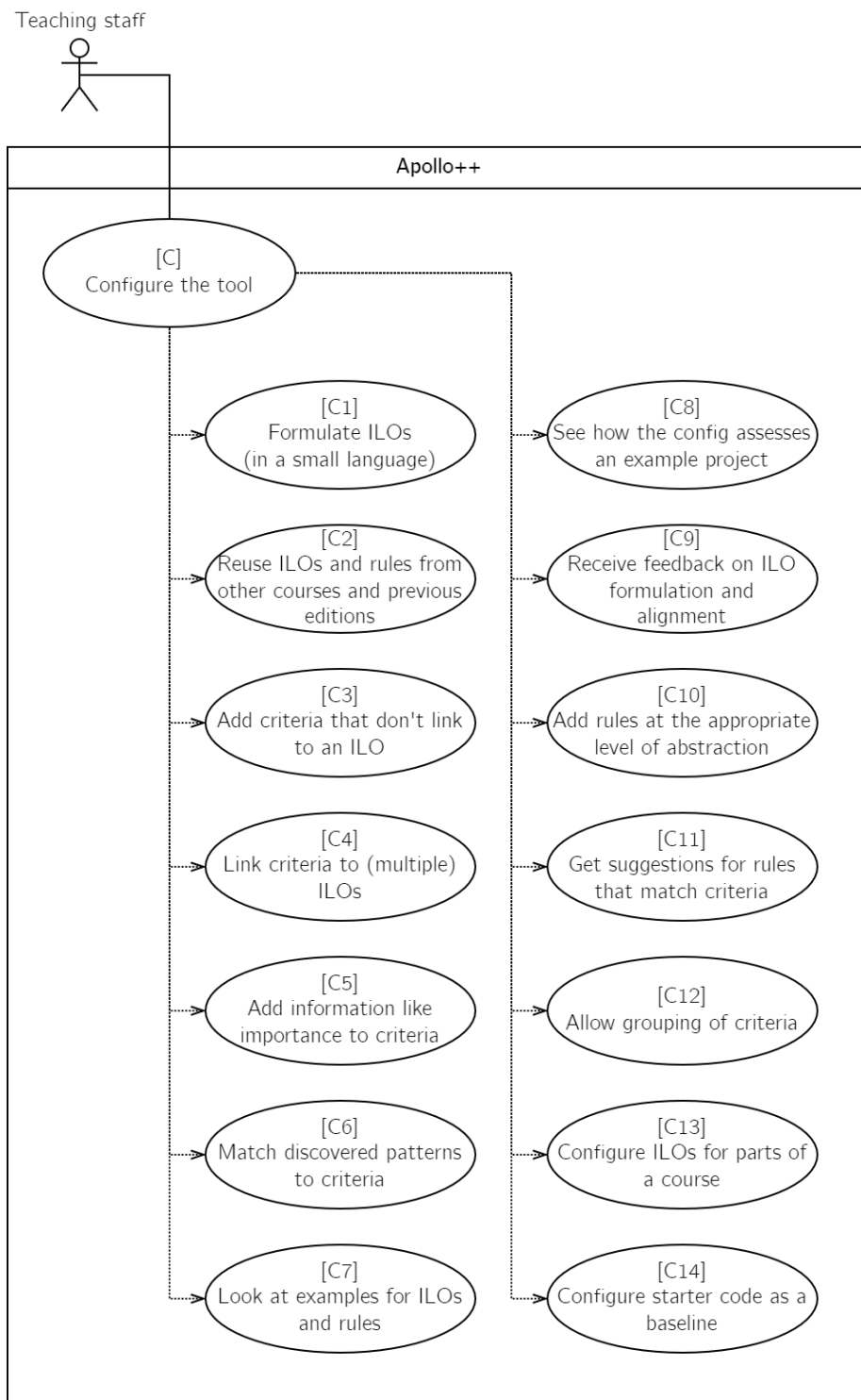


Figure 3.4: Concrete actions related to use case *C* and configuring the tool.

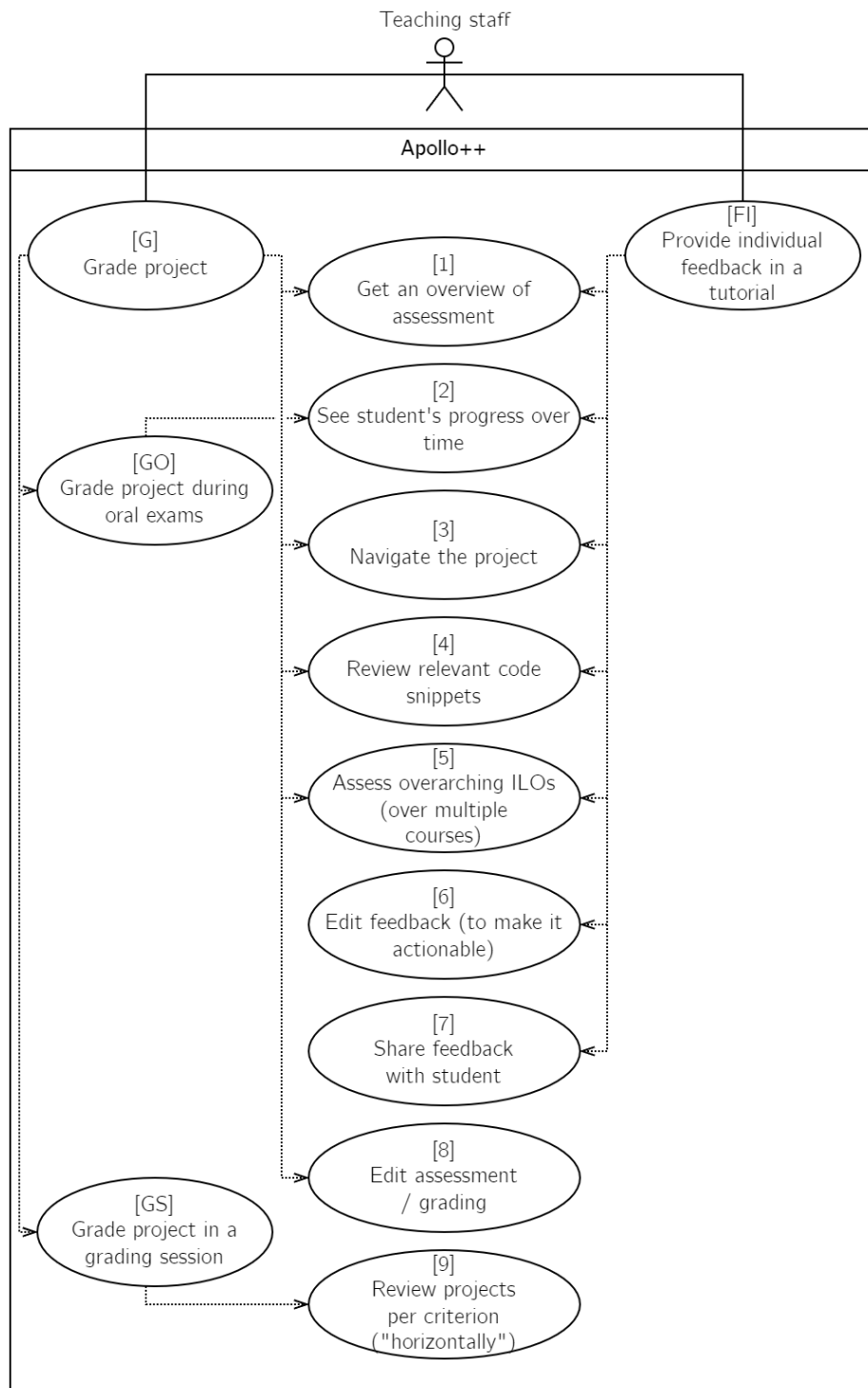


Figure 3.5: Concrete actions related to use cases *G* and *FI* about grading and providing individual feedback.

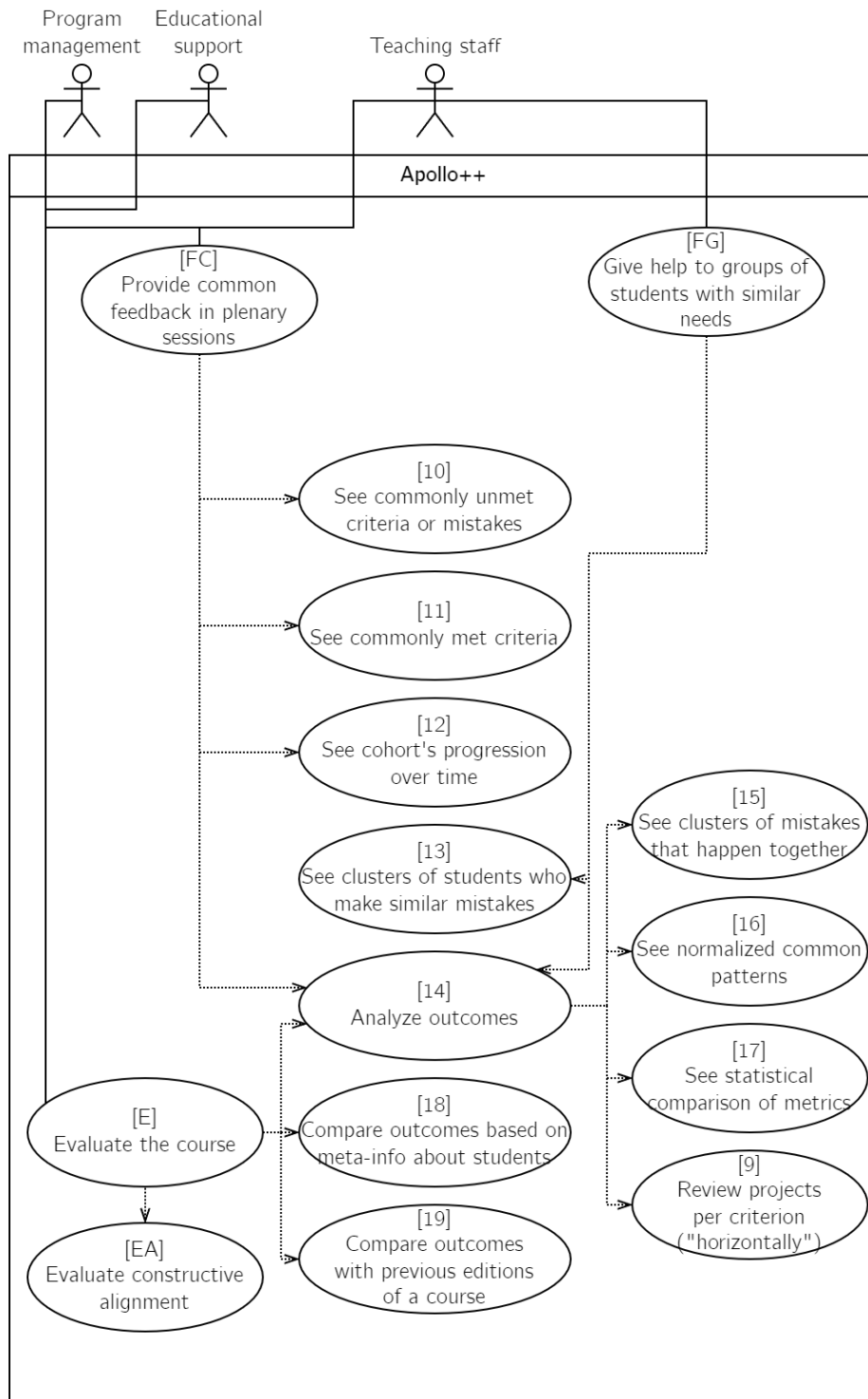


Figure 3.6: Concrete actions related to use cases *FC*, *FG* and *E* on plenary feedback, feedback for groups and course evaluation.

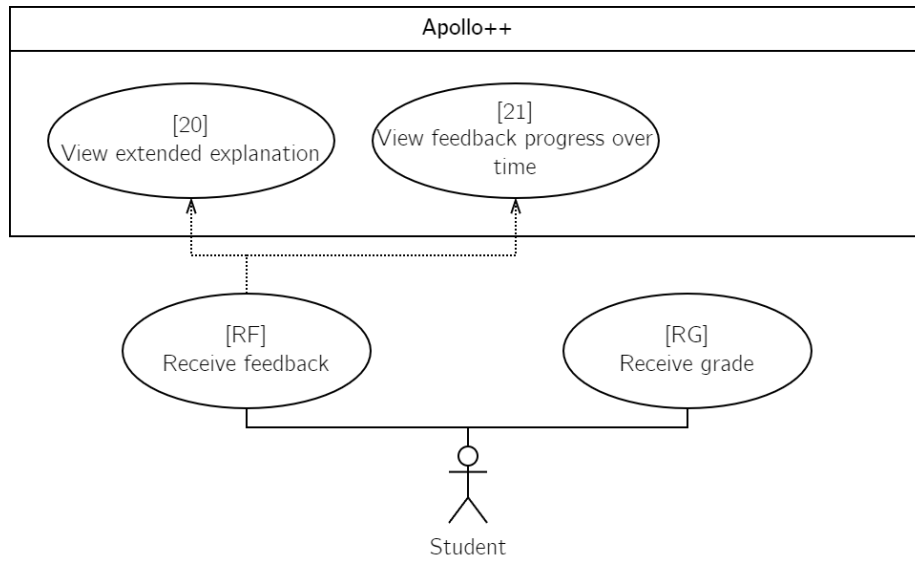


Figure 3.7: Concrete actions related to student use cases *RF* and *RG* about receiving feedback and grades.

system will be required for students to receive extended feedback and review their progress beyond what may be possible in external platforms.

3.5 Requirements

The use cases and concrete actions described in the previous section are derived from the full list of requirements, which can be found in table 3.1. Each requirement is linked to related use cases or concrete actions (or “General” if it concerns the system as a whole), to give this long list some structure. For further reference in this document, each requirement has a unique identifier, starting with *F* for functional requirements and *Q* for quality requirements. The table also shows the stakeholders that voiced concerns that led to this requirement and the number of participants that voiced these concerns during the interviews. The last column shows the chapters of this document in which each requirement is mentioned.

Table 3.1: Requirements based on the concerns voiced by stakeholders

ID	Use case	Stakeholder	Count	Requirement	Section
F1	1	Tutor in detail	1	Provides suggestions for where an assessor should look at the code	8
F2	1	Tutor in detail	3	Groups similar problems and repetitions of the same problem	8

ID	Use case	Stakeholder	Count	Requirement	Section
F3	1	Tutor in detail	4	Gives quick and clear overview of assessment	8
F4	1	Tutor in detail	1	Provides option to ignore all similar mistakes	7
F5	2, 21	Tutor in detail, Student	4	Shows a student's progression over multiple exercises	8
F6	4	Tutor in detail	3	Gives evidence for assessment with relevant code snippets	6
F7	5	Tutor in overview	1	Assesses overarching ILOs (e.g. security as a cross-cutting concern)	5
F8	6	Tutor in detail	1	Lets tutor extend an automatic comment with constructive feedback	4, 8
F9	6	Tutor in detail	3	Never shows feedback directly to students automatically	8
F10	7	Tutor in detail	2	Allows tutor to share feedback with students	8
F11	8	Coordinator, Examination board	3	Allows teachers to make their own assessment, e.g. in a rubric	4, 8
F12	8	Coordinator, Tutor in detail, Examination board	6	Does not give an (automatic) grade	6, 8
F13	9	Tutor in detail, Tutor in overview	3	Supports comparing projects by criteria, e.g. to grade projects per criterion (with anonymized submissions, for some cases)	8
F14	12	Tutor in overview	2	Shows progress of the cohort during the course	8
F15	13	Tutor in overview	1	Shows clusters of students making similar mistakes	8
F16	15	Tutor in overview	1	Shows clusters of mistakes that happen together	8
F17	15	Tutor in overview	1	Links to code examples for common mistakes	8
F18	16	Tutor in overview	1	Shows normalized structural issues / patterns	8

ID	Use case	Stakeholder	Count	Requirement	Section
F19	17	Tutor in overview	1	Compares metrics like execution times	8
F20	18	Tutor in overview	1	Compares groups based on meta-information (e.g. gender, pre-knowledge)	8
F21	19	Tutor in overview	1	Compares with previous editions of the course (retroactively, to test the results of interventions)	8
F22	20	Tutor in detail, Student	3	Provides insight into / extra explanation of feedback, e.g. by linking to terminology	8
F23	1, 3	Tutor in detail	3	Aids in understanding and navigating the program	8
F24	1, RF	Tutor in detail	1	Uses objective language for objective observations	8
F25	10, 11	Tutor in overview	3	Highlights ILOs which are commonly (not) achieved	8
F26	3, 4	Tutor in detail	2	Provides quick access to code context and details	5, 8
F27	4, 20	Coordinator	1	Provides explanation of assessment on demand	8
F28	BS, BR	Coordinator, Tutor in detail	5	Integrates with existing platforms, like Canvas, Atelier and GitHub	4, 8
F29	C	Configurator	1	Can separate parts of criteria that change (e.g. rules of the game) and parts that stay the same (e.g. overall structure)	
F30	C	Coordinator	1	Assess non-functional aspects	6
F31	C	Coordinator	1	Does not impose restrictions on the type of exercise	6
F32	C1	Configurator	1	Has a small language to formulate ILOs	8
F33	C10	Configurator	1	Supports rules about variables in camelCase	6
F34	C10	Configurator	1	Supports rules about if-statements	6
F35	C10	Configurator	1	Supports rules about overly long methods	6

ID	Use case	Stakeholder	Count	Requirement	Section
F36	C10	Configurator	1	Supports rules about consistent style	6
F37	C10	Configurator	1	Supports measuring relevant metrics (e.g. execution time)	6
F38	C10	Tutor in detail	2	Recognizes code patterns / design patterns	6
F39	C10	Tutor in detail	1	Reports code smells	6
F40	C10	Tutor in detail	1	Recognizes improvement in code quality	5, 8
F41	C10	Tutor in detail	1	Supports marking unused code	6
F42	C10	Tutor in detail	2	Assesses comments and code file headers	6
F43	C11	Configurator	1	Gives suggestions for rules	7
F44	C12	Configurator	1	Supports grouping of criteria	5
F45	C13	Configurator	2	Supports different ILOs for consecutive parts of a course	5
F46	C13, 1	Tutor in detail	2	Only shows learning outcomes relevant to the current part of the course	5
F47	C14	Coordinator	1	Works with exercises where students improve existing code	5
F48	C2	Configurator	1	Provides option to copy relevant parts from previous course editions	5
F49	C2	Configurator	5	Has a community/library to share ILOs, criteria and rules with others and other courses, with explanation and interpretation	8
F50	C3	Configurator	1	Allows criteria without link to an ILO (e.g. on grammar in a report)	5
F51	C4	Configurator	1	Can link criteria to multiple ILOs	5
F52	C5	Configurator	1	Can configure importance of criteria (to aid sorting)	5
F53	C6	Configurator	1	Allows configurator to match discovered patterns to criteria	7

ID	Use case	Stakeholder	Count	Requirement	Section
F54	C7, C8	Configurator	4	Gives support while configuring ILOs, criteria and rules, with documentation, examples and/or live assessment of example project	8
F55	C9	Configurator	1	Gives feedback on ILOs and suggests what can be assessed in the tool	7, 8
F56	C9	Configurator	1	Gives feedback on link between ILOs and criteria	7, 8
F57	E	Tutor in overview	1	Provides dashboard or report for evaluation at the end of the course	8
F58	EA	Examination board, Educational support	2	Can provide information about constructive alignment	8
F59	FI, RF, 7	Student	1	All feedback is checked by a teacher or TA	8
F60	General	Coordinator	1	Fits in the context of the module	6
F61	General	Coordinator, Tutor in detail	2	Supports the programming language(s) used in the course	6
F62	General	Tutor in detail, Examination board	2	Assessment of a criterion is not binary	6
F63	Out of scope	Coordinator, Tutor in detail, Examination board	3	Works with a plagiarism checker	
F64	Out of scope	Examination board	1	Helps in the inspection of borderline cases	
F65	Out of scope	Tutor in overview	1	Reports on consistency between assessors	
F66	Out of scope	Tutor in overview	1	Reports on consistency between assessors for a single project	
F67	RF	Student	1	Also notes things that are done well, not just mistakes	8

ID	Use case	Stakeholder	Count	Requirement	Section
F68	RF	Student	1	Is less hesitant in the wording than to tutors	
F69	RF	Student	1	Shows who (which tutor) shared the feedback	5, 8
F70	RF, RG	Student	1	Shows assessment as a rubric	
Q1	AL	Developer / Maintainer	1	Can easily be extended to support additional languages	6
Q2	AP	Developer / Maintainer	1	Can easily be extended with different metrics and types of patterns	6
Q3	BS, BR	Coordinator	1	Integrations with other systems are seamless (details are important here)	4
Q4	C	Configurator	1	Sensibly groups options in the configuration UI	8
Q5	C	Configurator	1	Hides unused options in the configuration UI	8
Q6	C	Coordinator	1	Provides basic functionality with minimal configuration	8
Q7	C	Coordinator	1	Rubrics are easy to import and export	
Q8	G, FI, FC	Coordinator	1	Is not overwhelming with 300 projects	8
Q9	General	Coordinator	1	In summative contexts, assessment has high reliability and validity	7
Q10	General	Coordinator	1	In formative contexts, assessment has sufficient reliability and validity to serve as a discussion starter	7
Q11	General	Coordinator	1	Works correctly and does not spew error messages	
Q12	General	Coordinator	1	Provides option to tweak every little setting	
Q13	General	Coordinator, Configurator	4	Is easy/simple/intuitive to use, especially on first interaction	8
Q14	General	Coordinator, Tutor in detail, Examination board	3	Assessment is reliable enough to be trusted	7

ID	Use case	Stakeholder	Count	Requirement	Section
Q15	General	Examination board, Educational support	2	Assessment is reliable, valid and transparent	7
Q16	General	Student	1	Has an understandable assessment process	7
Q17	General, 3	Tutor in detail	1	Has fast and easy interaction, quick navigation	8
Q18	RF	Student	2	Gives constructive feedback, or keeps silent otherwise	8

From the list of requirements, we can see that the most requested feature of our tool is that it will not automatically give a grade (F12). Six participants mentioned this during the interview, including coordinators, tutors in detail and members of the examination board. Next on the list is the integration with existing platforms (F28) and the ability to share course configuration, including ILOs, criteria and rules with others to reuse or as an example (F49).

We deemed four requirements to be out of scope for this project, namely that

- the tool includes or works with a plagiarism checker (F63). This is out of scope, because Apollo++ should integrate in other platforms, which often already come with tools for plagiarism checking or can integrate with those tools.
- the tool helps in the inspection of borderline cases when it comes to grades (F64). This is out of scope, because the tool does not perform automatic grading and thus has no ability to highlight borderline cases automatically.
- the tool can analyse the consistency between different assessors (F65, F66). This would require comparing manually entered assessments, which is beyond the scope of this project.

Based on these requirements, the next section describes the mission for this tool. In the coming chapters, we will assume that these requirements describe the intended behaviour of the system. Each statement about the design will be accompanied by a reference to one or more requirements, both to explain why that statement is needed and to show how all requirements are implemented.

3.6 Mission

Apollo++ aims to be *supporting* and *flexible*. It is *flexible*, because it supports every criterion that could be assessed by looking at code and every such criterion in a course is configurable in the tool. It integrates with LMSs and other tools and can be extended to support multiple programming languages. It is *supporting*, because it helps teachers assess projects by highlighting the code that is relevant to certain assessment criteria, rather than performing automatic

grading. The tool also leverages previous submissions to support teachers during the configuration phase with suggestions and immediate feedback.

Part II

Architecture

In part I, we determined the requirements for a tool that performs automated assessment on larger programming projects and formulated *support* and *flexibility* as the two guiding principles for our design decisions. In this part, we introduce a design for Apollo++ based on those requirements as an attempt to answer our third question: *How to build this tool?* In part III we will then evaluate this design.

The chapters in this part describe the design for Apollo++ from different viewpoints. Chapter 4 describes the components that make up this system, which gives an overview of the system as a whole and helps to place the other views in context. Next, chapter 5 describes how the data in the system is structured, chapter 6 describes how the assessment pipeline works, chapter 7 describes how the tool supports users in the configuration phase and chapter 8 describes how the user interface is organized to provide access to all functionality in relation to the interfaces provided by an existing system. Together, these views should provide a clear picture of the system and include the most important decisions for an implementation.

Chapter 4

Components

Given the aim of a *flexible* system, an architecture of loosely coupled components is an obvious choice. This primarily addresses the integration with multiple other systems (F28) and enabling these integrations to be as seamless as possible (Q3). Each component, of course, also addresses its own concerns, which are described below.

This decomposition is visualised using a UML component diagram, with a few extensions: a dashed line is used to indicate a direct correspondence between a back-end and front-end component, which need no other interfaces than to communicate between each other; and incoming and outgoing interfaces are not all connected like they would normally be. Interfaces with the same labels should be read as if they are connected to the corresponding interfaces on other components. Communication between the components is visualised with a UML activity diagram.

Figure 4.1 shows the components of Apollo++. The diagram shows three types of components, namely the *Integrator* component on the left, which connects the tool to other systems; the core components in the centre, which provide the backend functionality; and the UI components at the right, which provide different parts of the user interface. Overall, this component decomposition allows us to integrate with different external systems by providing multiple *Integrator* components for systems like Canvas, Atelier or GitHub (F28). Because the UI is split in different components, it allows parts of the UI to be integrated in an external system if that system supports such integrations. This helps to make the integration as seamless as possible (Q3).

All components play a role in fulfilling one or more of the use cases described in figure 3.3:

- The configuration (UC C) is handled in the *Assessment Configuration* and *Configuration UI* components.
- External tools can submit code and retrieve the assessment (UC BS, BR) through the *Integrator* component.
- Grading and providing individual feedback (UC G, GO, GS, FI) happen through the *Integrator* component where possible and the *Code Review UI*

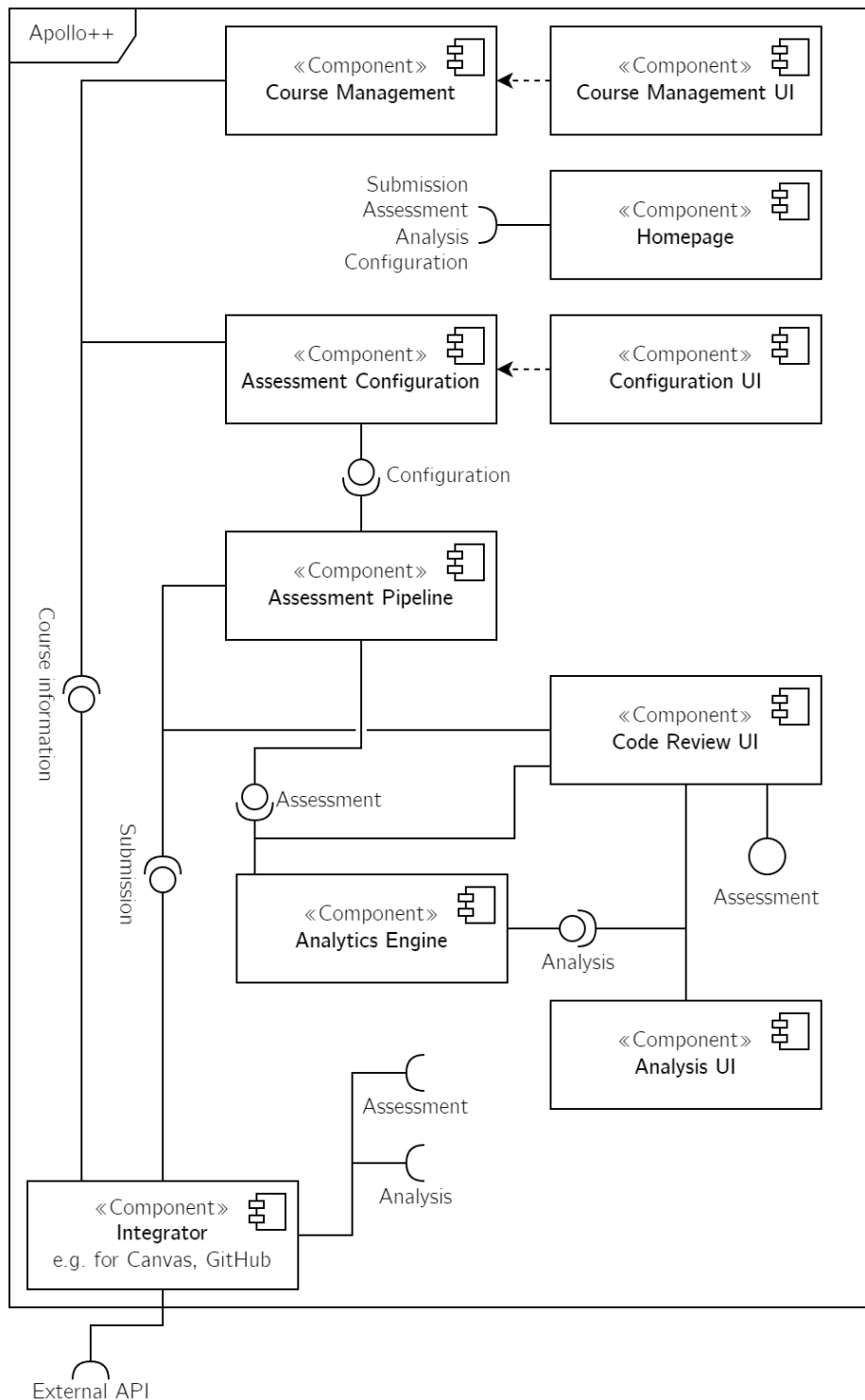


Figure 4.1: Component decomposition of Apollo++, with three types of components: the *Integrator* connects with other systems, the core components in the centre provide the backend functionality and the UI components on the right provide the user interface.

where it is not, backed by the *Assessment Pipeline*. The integrated system or the internal UI allow users to view the automated assessment (UC 1, 5), navigate the project (UC 3), review relevant code snippets (UC 4), edit feedback and assessment or grading (UC 6, 8) and share the feedback with students (UC 7). The *Assessment Pipeline* provides the automated assessment data (UC 1, 4, 5).

- Students receive feedback and/or grades (UC RF, RG) through the *Integrator* in an external system and/or in the *Code Review UI*.
- Feedback for groups of students (UC FG, FG) and course evaluation (UC E, EA) happen through the *Analysis UI*, which is backed by the *Analytics Engine*. The latter performs all analysis that spans multiple submissions, over time or by multiple students. This covers all actions that are related to the mentioned use cases, but also includes the progress over time of a single student (UC 2) and reviewing projects “horizontally” per criterion (UC 9).
- The *Course Management*, *Course Management UI* and the *Homepage* are necessary for all use cases. The tool needs to be linked to a course in an external system for any functionality to work. If the external system does not allow for deep linking, then the *Homepage* provides access to all UIs within the tool.

To clarify how these components communicate, the activity diagram in figure 4.2 shows how the components interact when a student uploads a submission, which gets automatically assessed and then manually reviewed by a teacher. (Also see figure 8.2 in chapter 8 for a more user-centric version of this activity diagram.) As soon as an (automated) assessment is available, this is pushed back to integrated systems via the *Integrator* component and when a tutor updates the assessment this is again propagated through the system. This allows teaching staff to extend the automated assessment or include their own feedback (F8, F11), while integrating with an external system as seamless as possible (Q3).

Given this reactive behaviour in the communication between components and the need to work with different types of *Integrator* components, a central messaging bus should be used for this communication. This helps to keep track of updates to the data and propagate it throughout the system, while also aiding in building new *Integrators* without changing all components that produce data.

Overall, the most important feature of this component decomposition is the ability to include different *Integrator* components to integrate with multiple platforms without changing the rest of the application. The reactive nature of communication between components allows for a combination of automated and manual assessment, while keeping all parts of the application and integrated systems up-to-date. The next chapter describes the structure of data at the interfaces between these components.

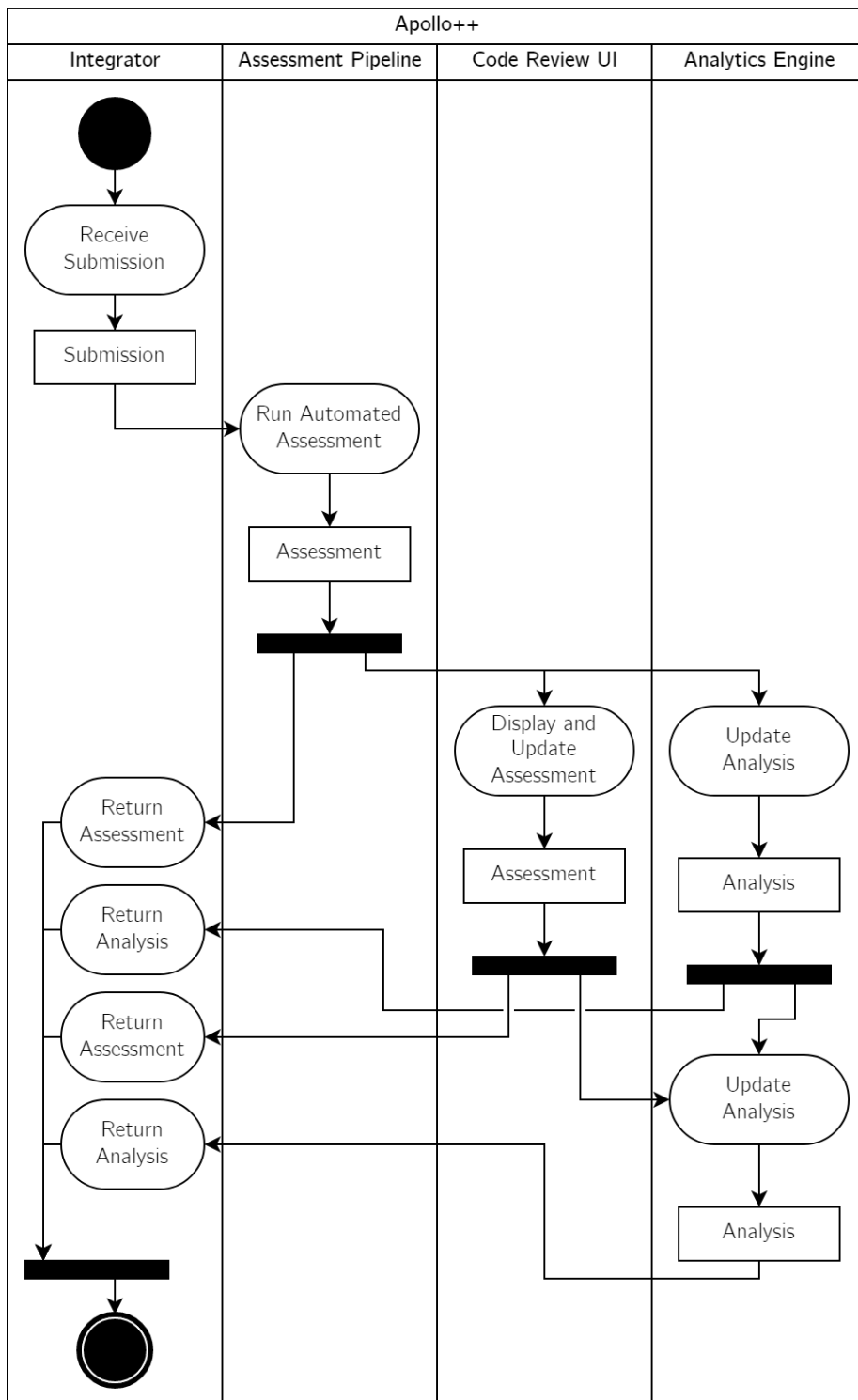


Figure 4.2: Activity diagram showing communication between components when a student submits their project and a tutor assesses it. The rounded boxes represent actions within a component, while the rectangular boxes indicate the type of data passed between components.

Chapter 5

Datamodel

This chapter introduces the datamodel for Apollo++: it describes the information that is shared between components when it comes to configuration, submissions and assessments. This model does not include the information around the analysis part of the system, as that is concerned with relatively standard analytical information and does not require much elaboration. The datamodel for configuration and results has some special features to meet the requirements of the tool, so it is worth discussing. The datamodel is presented as a UML class diagram, but the diagram is not exhaustive. Standard features like titles, descriptions, names and dates are omitted to focus on the structure of the model.

Figure 5.1 shows the datamodel for Apollo++. The central part of the system is the configuration: ILOs, criteria and rules. ILOs can be linked to many criteria, but a criterion does not necessarily link back to an ILO. This may be the case when grammar in a report is assessed but “the student can write grammatically correct reports”¹ is not an intended outcome of the course (F50). A criterion could also apply to multiple ILOs (F51). Criteria are automatically assessed through rules configured by a teacher. These rules can be based on code patterns or based on metrics, which will be explained in chapter 6. Criteria can be related to multiple rules, which can be marked as providing positive or negative examples for meeting a criterion.

Courses come with a set of ILOs, but can also include those criteria that are not linked to an ILO. To provide more organization, assignments can be created, which can reference their own set of ILOs and criteria. This allows the configurator to indicate which ILOs are relevant at each point in the course (F45), so that the tool will only show those ILOs and criteria when assessing a submission for that assignment (F46).

Metadata around a criterion in a course can optionally be added through a separate data structure. This allows grouping of criteria by category to create an organization similar to a rubric or assessment form (F44) and an importance, which can be used to highlight important criteria when assessing a submission (F52). Note that the cardinalities round ILOs, criteria, courses and assignments

¹Note that Apollo++ will only automatically assess code, but ILOs and criteria around reports could still be added in the system.

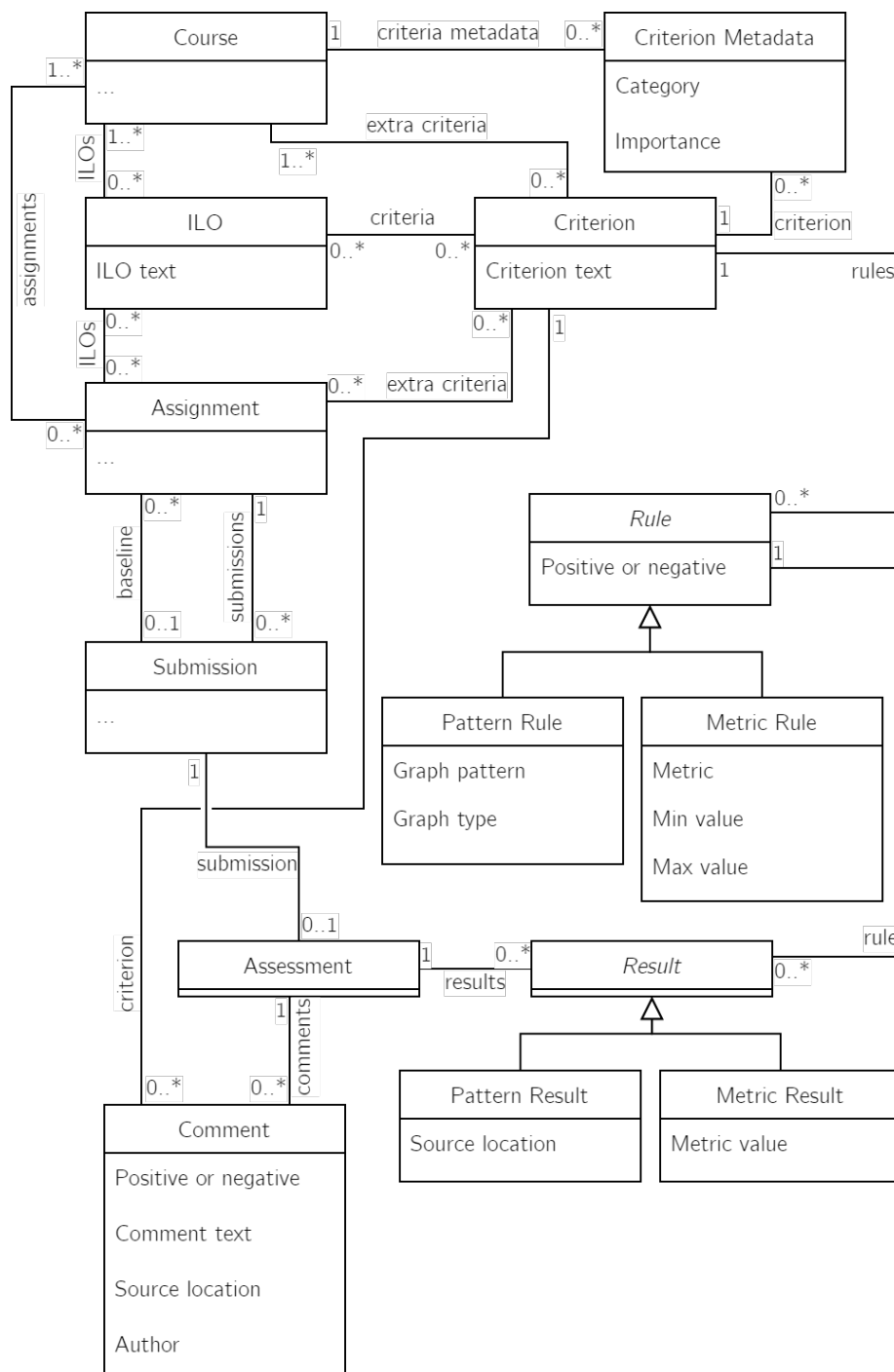


Figure 5.1: Datamodel for configuration, submissions and assessments in Apollo++. The top part describes the structure of courses, assignments, ILOs and criteria. Criteria are linked to rules, described in the centre-right. Submissions relate to an assignment and receive an assessment, which includes comments and results that are based on a rule.

are very flexible to allow reusing ILOs (and their related criteria and rules) across multiple courses (F7, F49) or reuse them from previous editions (F48). Assignments also include a special reference to a submission, which enables it to be used as a baseline on which students can build or improve (F47). The assessment results for this submission can be compared to new submissions to see how students improved the code using the analysis features (F40, described in chapter 8). Besides the hard link to one or more courses, Apollo++ poses no restrictions on the type of assignment given, as long as it involves submitting an artefact (F31). With this flexible structure, the tool should be suitable for use in any course where the assessment approach makes sense (F60).

Each submission can get a single assessment, which combines automatic assessment results and comments written by a human. The results link back to the rule that produced the result, which in turn links it to a criterion. Pattern results have a reference to the source code location (F26) and metric results include the value of the metric that triggered the result. Manual comments also reference a criterion and are marked as positive or negative examples, so they can be included in the same overviews as the automated assessments. They include a reference to the relevant location in source code and who authored the comment (F69).

This datamodel describes how the data that is passed between components relates and what information is available in the application. The next chapter describes how the rules in this model are used to give assessment results for a submission.

Chapter 6

Assessment pipeline

In this chapter we discuss the inner workings of the *Assessment Pipeline* component, which takes in the student’s code and outputs the assessment results based on a configuration, addressing requirements about the assessment process. The pipeline is visualized using a “pipeline diagram”, which is explained in figure 6.1. The square in the centre represents a component in the pipeline, which is similar to a process step in flowcharts. The slanted boxes represent different types of information flowing through the pipeline and the curved box represents stored information that is used in the process. The difference between pipeline input and stored information is that the pipeline input varies for every run of the pipeline, whereas the stored information stays the same.

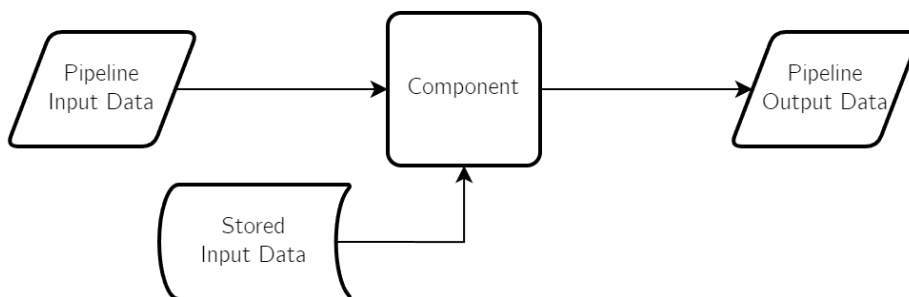


Figure 6.1: Example of a pipeline diagram. The square box represents a component, the slanted boxes represent types of information flowing through the pipeline and the curved box shows stored information that remains the same across runs.

Figure 6.2 shows the assessment pipeline for Apollo++. The pipeline starts at the top with some input data, which is the code a student submitted in a certain programming language. Boxes with the `<lang>` marker are specific for a certain programming language, so these components need to be implemented when adding a new programming language to the system. The pipeline only contains two components that are language-specific, so the changes needed to support another language are contained in those modules (Q1, F61).

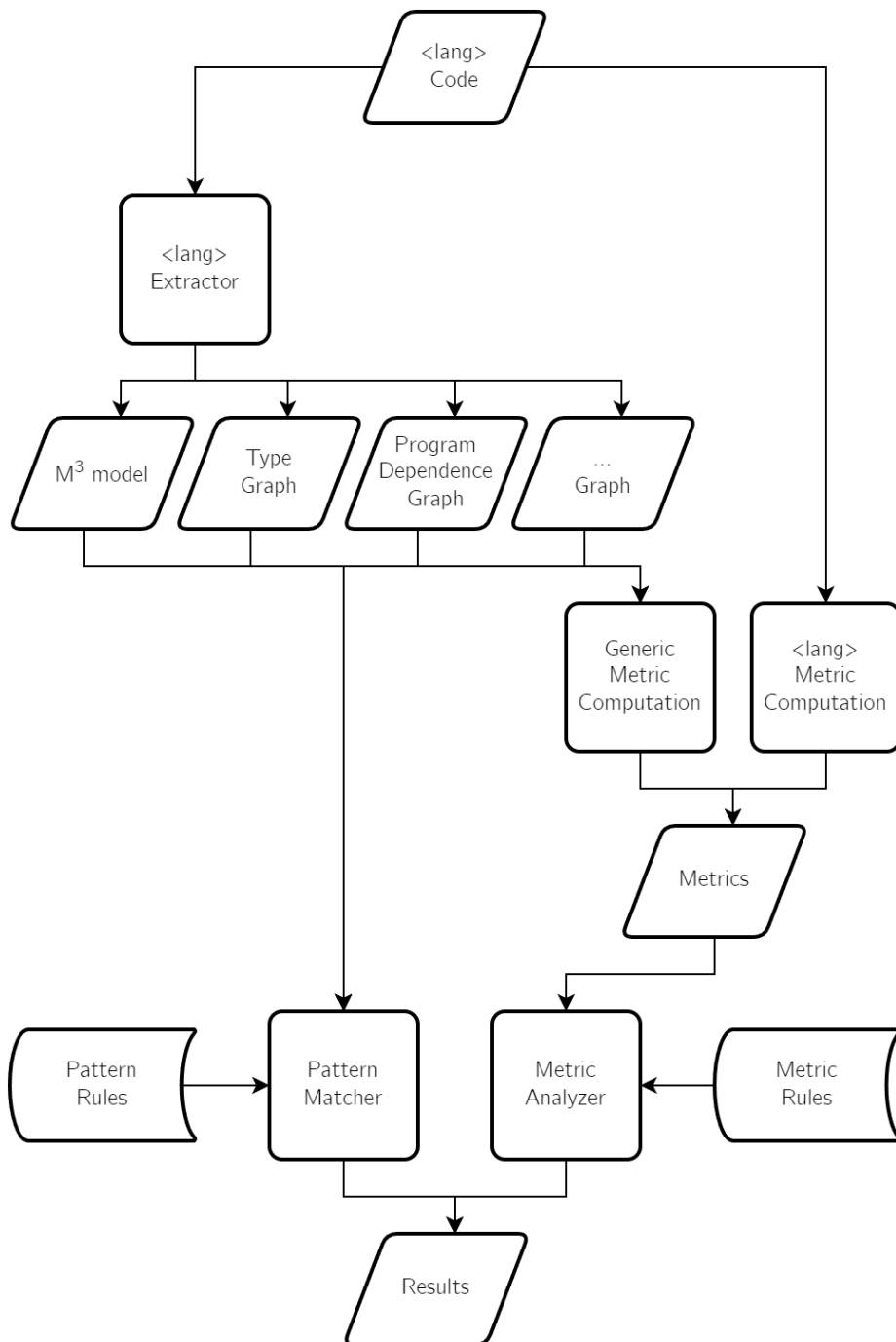


Figure 6.2: Pipeline diagram of the Apollo++ assessment pipeline. The slanted box at the top represents the input data (code in a certain programming language), which flows through the pipeline. Boxes marked with `<lang>` are specific to a certain programming language and all others are generic components.

The first component in the pipeline is the *Extractor*, which takes the source code and translates it into different models. This *Extractor* should be based on a compiler or editor API where possible to prevent reimplementing the full compiler with possibly slight changes in semantics. The first model provided by the extractor is the M^3 model described by Basten et al. (2015). This model defines a set of relations that describe a program, such as the containment of methods in classes, the reference of a type in a field declaration etc. This makes it a good candidate for a universal representation which can be used to compute metrics by the *Generic Metric Computation* component. The other models created by the *Extractor* are graphs, such as a graph with type relations (inheritance, references, containment, see section 6.1), a program dependence graph (PDG) or other graph representations.

New graph representations can be added here to support specialised features in a programming language or a type of pattern that is hard to define using the other representations. This flexibility allows the tool to be extended with new types of patterns by extending an *Extractor* to produce this new type of graph (Q2). All graphs are passed on to a universal *Pattern Matcher*, which takes a graph and matches the stored pattern rules configured by a teacher against that graph. In the simplest form this is just subgraph matching: checking if a pattern graph is contained in the extracted program graph, but more complex situations may require extensions such as checking for the absence of nodes or edges. In section 6.2 we describe a structure that also enables patterns that forbid certain elements.

By using a generic component for this task, rules at different levels of abstraction can be implemented with different graphs while maintaining a common infrastructure for all rules (F33 to F36, F39, F41, F42). On an appropriate graph, these rules could also be used to recognize design patterns (F38). The *Pattern Rules* are configured by teachers and provided as subgraphs with a relation to some assessment criterion, such that the matches can be linked to a criterion in the *Results*.

The top half of the pipeline shows how metrics are handled (F37). A *Generic Metric Computation* component takes the M^3 model generated by the *Extractor* to compute common metrics based on the relations in that model. A language specific *Metric Computation* is used to calculate metrics that require access to the source code or AST and thus are only applicable to one programming language. These *Metrics* are combined and passed to the *Metric Analyser* which uses *Metric Rules* to make conclusions about criteria related to those rules. These rules can be used to define ranges in which a metric should fall to meet a criterion, for example. The combination of graph matching rules and metric rules covers most types of criteria, including non-functional aspects of the code (F30).

The *Results* are a list of criteria with examples and counterexamples found in the code as evidence that a criterion has been met or not (F6), but without a yes or no decision on meeting the criterion (F62). Additionally, this includes pointers to parts of the program that were not used in examples or counterexamples, to help assessors find the blind spots of the tool and evaluate those parts manually. A grade or other summative conclusion is not included in these results (F12).

6.1 Type graph

Besides common program representations, such as a control flow graph or program dependence graph, we propose a graph that represents the class structure of an object-oriented program. The nodes in this graph represent the classes, interfaces, primitive types, fields, constructors, methods, parameters and variables in the program, with directed labelled multi-edges indicating the relationships between these elements. For classes this includes *extends* and *implements* indicating that the class extends the class or implements the interface the edge points to; an *invoke* edge means that the element invokes the method pointed to, *overrides* indicates a method overriding a method on a base class and *accessesField* indicates that an element reads or writes a field. Any element that uses a type in some way has an edge to indicate that it *dependsOn* that type: fields use the edge to indicate their type, methods point to their return type and parameter types etc. Finally, the *contains* relation means that one element contains another: classes contain their fields and methods, a method contains its parameters etc.

The *invokes*, *accessesField* and *dependsOn* relations are extended to all elements that contain the element with the relation. So if `methodA` in `classA` invokes `methodB`, then `classA` also gets an *invokes* edge to `methodB`. This makes it easier to define patterns without having to include every element.

The nodes of the type graph can also have one or more labels, which can be encoded as labelled self-edges. These include either *inProjectDecl* for elements that are declared within the project or *externalDecl* for elements from the standard library or other external libraries. Elements with *inProjectDecl* also include a *scheme*: the type of element that node represents, e.g. a class or field. Elements with *externalDecl* include the scheme as well as the fully qualified name of that element. Lastly, the elements can be labelled with a *modifier* like public or private and a *nameClass*, which means that the name of the element includes a certain substring. The latter can be useful to detect patterns with common names, such as a `*Factory` or `*Visitor`.

6.2 Pattern rules

Some criteria require checking for the absence of certain nodes or labels, rather than a simple subgraph check. Take for example the criterion “All fields except constants are private.” This should positively match all fields that have a `private` modifier but no `final` modifier (in Java), and negatively match all fields that do not have either modifier. This is called a negative application condition, which is commonly used in graph transformation techniques (Heckel, 2006). Figure 6.3 shows two patterns, which both match a single field in the TypeGraph and have a forbidden node-label indicated by an exclamation mark and red colour. The left pattern is a positive match on a field that has a `private` modifier and no `final` modifier, whereas the right pattern is a negative match on a field that does not have either a `private` or a `final` modifier.

This can be implemented by first finding all matching subgraphs for the pattern without any of the forbidden labels and then discarding all mappings that can be extended to include one of the forbidden elements. One drawback of this approach is that it is more difficult to extract forbidden elements from code.

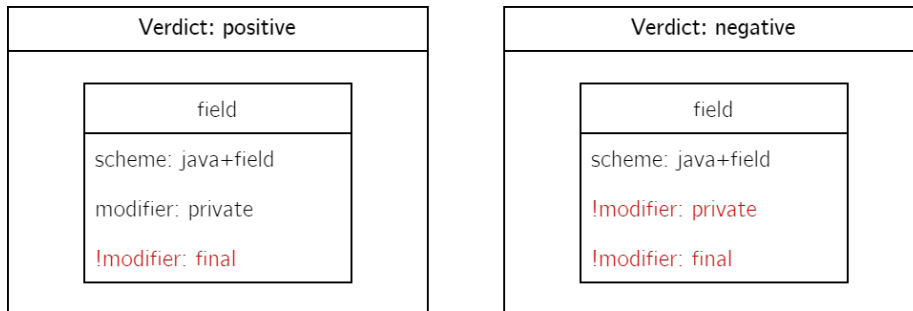


Figure 6.3: Two patterns for “All fields except constants are private.” using negative application conditions. The left pattern positively matches fields that are marked as private, whereas the right pattern negatively matches fields that are neither private nor final.

Instead, we propose a structure that also enables patterns to forbid certain elements, but also makes it easier to create variations of a pattern and to grow the pattern with nodes that are connected in previously submitted solutions (see chapter 7).

Figure 6.4 shows a tree of patterns, where each child extends its parent. This notation is convenient for a case with many variations, because the base pattern is not repeated in every pattern. Figure 6.5 shows how the patterns are combined into a full pattern at each node of the tree.

Rather than only *positive* or *negative* matches, there are also *neutral* matches to ignore certain conditions. The final verdict is given by the deepest matching patterns in the tree. In this example, all fields will match with the root of the tree (*a*). Then we recursively try to extend the match with each child. If none of the children match, then this is our final match. If one or more of the children return a match, then we continue searching in those branches and discard the current node. We can consider four cases that match the root pattern (*a*):

- A **non-private**, **non-final** field. Neither of the children (*b*) and (*c*) can extend the mapping, so we return the mapping at (*a*) with the verdict *negative*.
- A **private**, **non-final** field. The mapping can be extended by (*b*), so we discard the result from (*a*). At (*b*), we recursively try to extend the mapping with the only child (*d*), which does not extend the mapping, so we return the mapping at (*b*) with verdict *positive*. The pattern at (*c*) can not extend the mapping, so that branch is discarded.
- A **final**, **non-private** field. The mapping can be extended by (*c*), so we discard the result from (*a*). (*c*) has no children, so we return the mapping at (*c*) with the verdict *neutral*. The pattern at (*b*) can not extend the mapping, so that branch is discarded.
- A **private final** field. The mapping can be extended by both (*b*) and (*c*), so we discard the result from (*a*). (*c*) has no children, so we return the mapping at (*c*) with the verdict *neutral*. At (*b*), we can extend the mapping with (*d*), so the result from (*b*) is discarded. (*d*) has no children, so we also return the mapping at (*d*) with the verdict *neutral*.

Note that because the tree has multiple branches, multiple results can be returned. In this case, the result is equivalent to the patterns with negative application conditions in figure 6.3: **non-private**, **non-final** fields are marked negative; **private**, **non-final** fields are marked positive; and other variations are neutral, which we consider to be the same as no match.

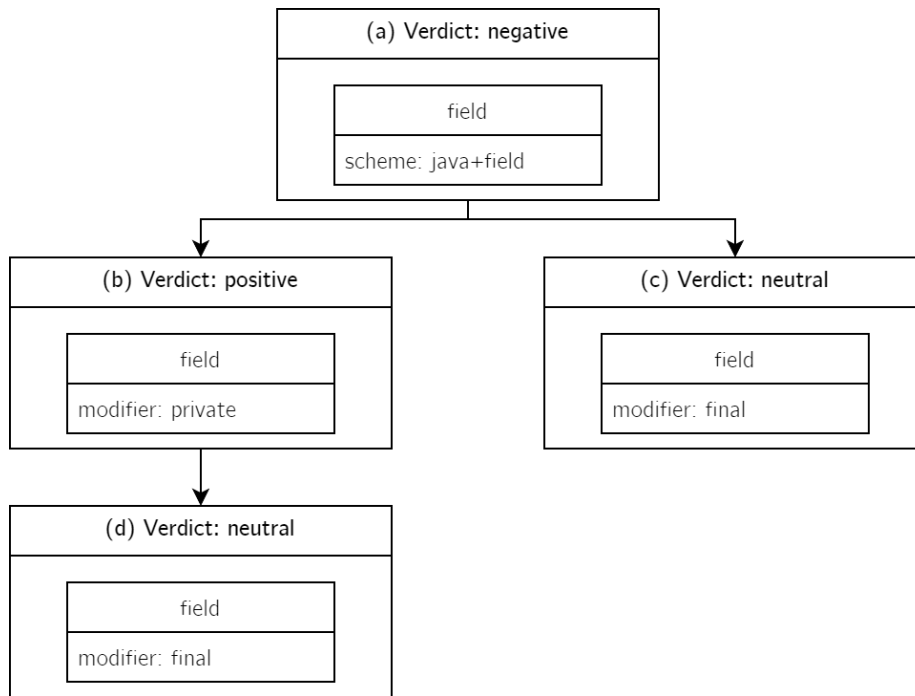


Figure 6.4: A pattern tree for “All fields except constants are private.” with the same semantics as the two patterns in figure 6.3. The top node negatively matches fields that are not private or final, the left node on the second level positively matches private fields, the node on the third level discards fields that are private and final and the right node on the second level discards nodes that are final.

Note that this process requires repeated subgraph matching of similar query graphs. To do so efficiently, the subgraph matching algorithm should include a way to extend a mapping found for a subgraph of the query into a full mapping for the entire query. Most subgraph matching algorithms work by recursively extending a mapping and building a search tree of all possibilities, so this should be possible to do with most algorithms. It is important to note, however, that the extension of a query may invalidate what was a valid mapping for a subgraph of that query. This is easily solved by checking the validity of the existing mapping against the subgraph of the query induced by the nodes of the original subgraph.

The benefits of this pattern tree structure are that small variations on large patterns can be described succinctly and that many variations can be checked efficiently by reusing the mappings found for parent nodes in the graph. The configuration engine can suggest extensions to the tree based on common neighbours that were found for each pattern in previous submissions. This and other

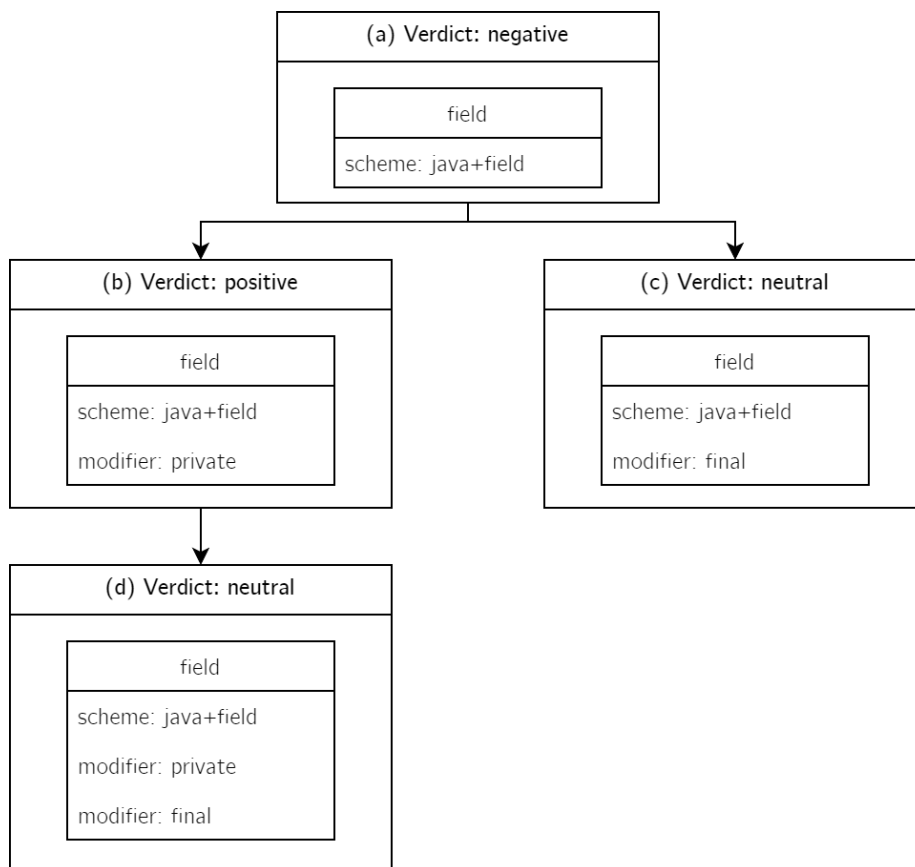


Figure 6.5: The pattern tree from figure 6.4 with each pattern expanded to the full subgraph.

functions of the configuration component are described in the next chapter.

Chapter 7

Assessment configuration

This chapter gives a functional decomposition of the *Assessment Configuration* component. The user that configures the tool uses the *Configuration UI* component to build the configuration as described in chapter 5. The *Assessment Configuration* component provides services to support the user during this process and which are, of course, surfaced in the *Configuration UI*. The different functionalities are visualised using a functional decomposition diagram.

Figure 7.1 shows the three main areas of functionality provided by the *Assessment Configuration* component: discovery of graph patterns (F53), providing suggestions for rules (F43) and evaluating constructive alignment (F55, F56).

Graph patterns to be used in rules can be discovered from old submissions. This helps teachers in the configuration process by surfacing common patterns in students code which can be positive or negative examples for certain assessment criteria. As described in chapter 6, Apollo++ uses different graph representations of a program to assess different types of criteria. These same graphs can be made of a large set of past submissions in order to find common graph patterns using well-known subgraph mining techniques (e.g. (Yan & Han, 2002)), but previous work (Mens et al., 2021) and some experiments showed that configuring these algorithms to find useful graphs is rather fiddly. Instead, we propose to use a large language model (LLM) to suggest starting patterns based on the written criteria, by simply providing the criterion and asking it for a graph pattern that would match code that fits with that criterion. This way we can harness the power of these models, but still provide a transparent and reliable assessment, because a teacher is still in charge of the actual rules that are used in the assessment process (Q9, Q10, Q14, Q15, Q16). These patterns are then matched against the set of old submissions and extensions and variations are suggested based on the neighbourhood of the matched subgraph in those submissions.

While using the tool, examples may pop up where the tool made an incorrect assessment. By highlighting these cases, the tool can provide suggestions to change graph patterns to exclude these cases from the positive examples and create a new rule to highlight the case as a negative example (F4). This can be done by extending the subgraph with elements found in the new situation, using the existing set of already assessed submissions for cross-checking.

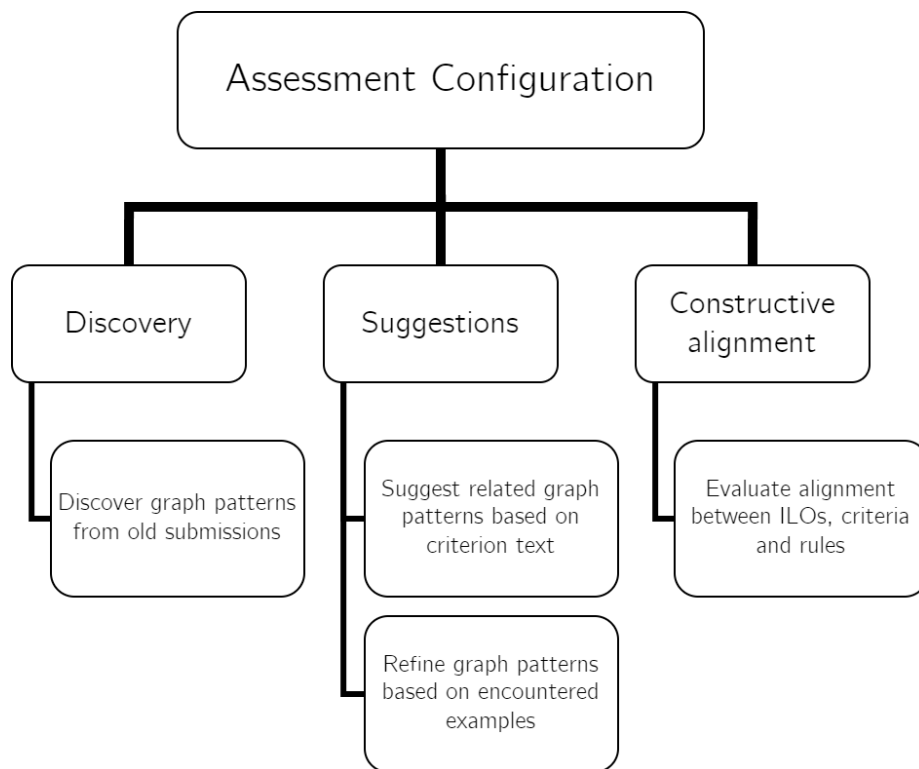


Figure 7.1: Functional decomposition of the *Assessment Configuration* component, showing the three main areas of functionality and the specific functions in that area.

Finally, the tool should give some feedback on the constructive alignment between ILOs, criteria and rules. Ideally the assessment criteria should cover all ILOs, but it can be difficult to keep track of this when creating an assessment form. Since the tool requests teachers to link their criteria to an ILO, it can easily provide feedback on this alignment during the configuration process (F56). It is also important to see what is assessed automatically by Apollo++ and which criteria are not covered by assessment rules. This can either be used to find which criteria should be configured with more rules, but can also serve as a reminder that manual assessment of these aspects remains important (F55).

This component is relatively simple and only needs to provide interfaces to execute the described functions from other components, which are mostly in the *Configuration UI* component. In the next section, we describe how this and all other information is available to users in the UI and how they interact with these and other features.

Chapter 8

User interface

This chapter discusses the user interface of Apollo++: the workflows and what information is available where in the UI. This addresses concerns from many direct users, because it describes how they interact with the tool. We use a UML activity diagram to visualise the typical workflow and a “navigation and information diagram” to describe how the UI of the tool is organised.

Figure 8.1 shows how a navigation and information diagram is organised: the system contains multiple UI components or pages, each represented by a box with the name of that component. These components represent pages within a website or sections of an application, but in the abstract represent a grouping of similar information that is commonly accessed together. The list of elements below the divider describes the information that is available through that component, either to be viewed, or to be edited. Arrows describe navigation actions (links, in the case of a webpage). These links can point to a component or page in general if it points to the header or to a specific bit of information if it points to that item. Similarly, links can originate from the general page, in which case you could expect to find them in a header or menu, or from a specific item, for example to link items in a collection view to a page with more detail about one element. Arrows are drawn in the direction of a typical navigation flow, but navigation in the reverse direction should also be possible.

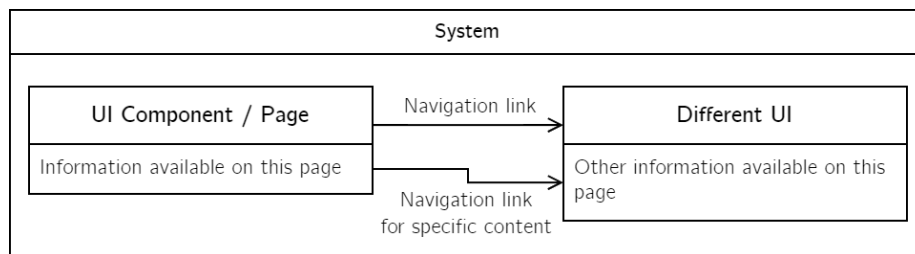


Figure 8.1: Schema of a navigation and information diagram. The boxes represent UI components or pages, listing the information that is shown. Navigation links are shown as arrows: between the headers for general page navigation and between specific lines of information for deep links.

Figure 8.2 shows the workflow of a student uploading a submission, which gets automatically assessed and reviewed by a tutor after which the student can review the feedback. After automated assessment takes place, the assessment is shared with the external system, but should not yet be visible to the student (F9, F59). If the external system does not support such access-control, then this assessment will not be available there. In this example the tutor uses the Apollo++ UI to review the assessment, update it and share it with the student, after which the updated assessment is again forwarded to the external system where the student is able to review the feedback (F10). This process helps to ensure that only constructive feedback reaches the student (Q18), which often requires more context than is viable to produce automatically in large projects.

Alternative flows include that the tutor first reviews the assessment in the external system and shares it with the student from there or that the students want to see more detail than is available in the external system and opens the *Code Review UI* of Apollo++ to review the feedback.

Figure 8.3 shows the navigation and information diagram for Apollo++ integrated with an external system like an LMS. The model for the external system is, of course, not complete, but includes only the components that are relevant when integrating Apollo++. The links between these systems show how Apollo++ integrates in the UI of an external system (F28). How much and which information can be displayed in there differs: when integrating through Learning Tools Interoperability (LTI) Assignment and Grade Services (AGS) (*Learning Tools Interoperability (LTI) Assignment and Grade Services Specification*, 2019), for example, assessment information is limited to a grade and a comment; the Canvas LMS additionally allows embedding an iframe in its SpeedGrader to directly access an LTI tool; Atelier and GitHub on the other hand are more focused on code and support inline code comments, but have no native notion of a grade. In all cases, there should be the option to link to Apollo++'s *Code Review UI*, so that all information can be accessed.

To make integrations as seamless as possible (Q3), as much information as possible should be shown in the native UI of the external system, with links to Apollo++ to get more details. If the system allows embedding, Apollo++ UI components can be embedded in the relevant places. Custom CSS could be used for each integration to match the style of the platform and provide a seamless integration. If neither is possible, links to Apollo++ should be added. The tool also has its own *Course Homepage*, to ensure all other components are accessible even if the external system is unable to link there. The *Course Management UI* is necessary to set up the tool, create courses and link it to the external system. The only requirement for these pages is that they have clear structure and search functionality, so that it remains usable with many submissions (Q8). Other than these, the components are typical for any such system and do not need further elaboration.

The most important part of the system for regular users is the *Code Review UI*, used for grading and individual feedback by tutors (UC G, FI) and receiving feedback by students (UC RF). This component should provide a great UI for assessing code projects, even if no automated assessment were available. The automated assessment is provided as assistance in this UI, so that tutors are in charge of their feedback and assessment (F8 to F12). This also makes sure that

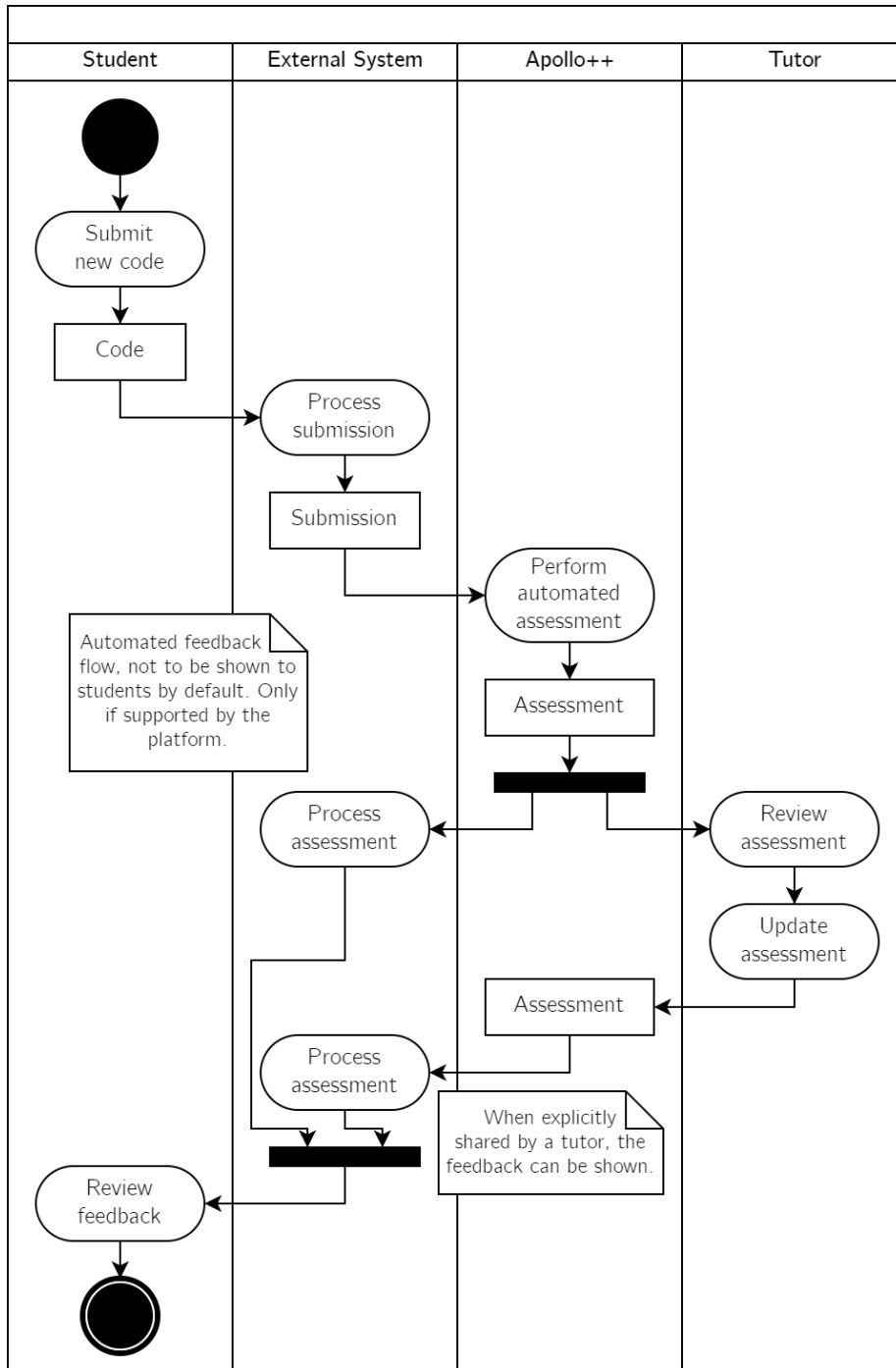


Figure 8.2: Activity diagram showing the actions taken when a student submits their project and a tutor assesses it. Rounded boxes represent actions and the rectangular boxes represent the data that is created.

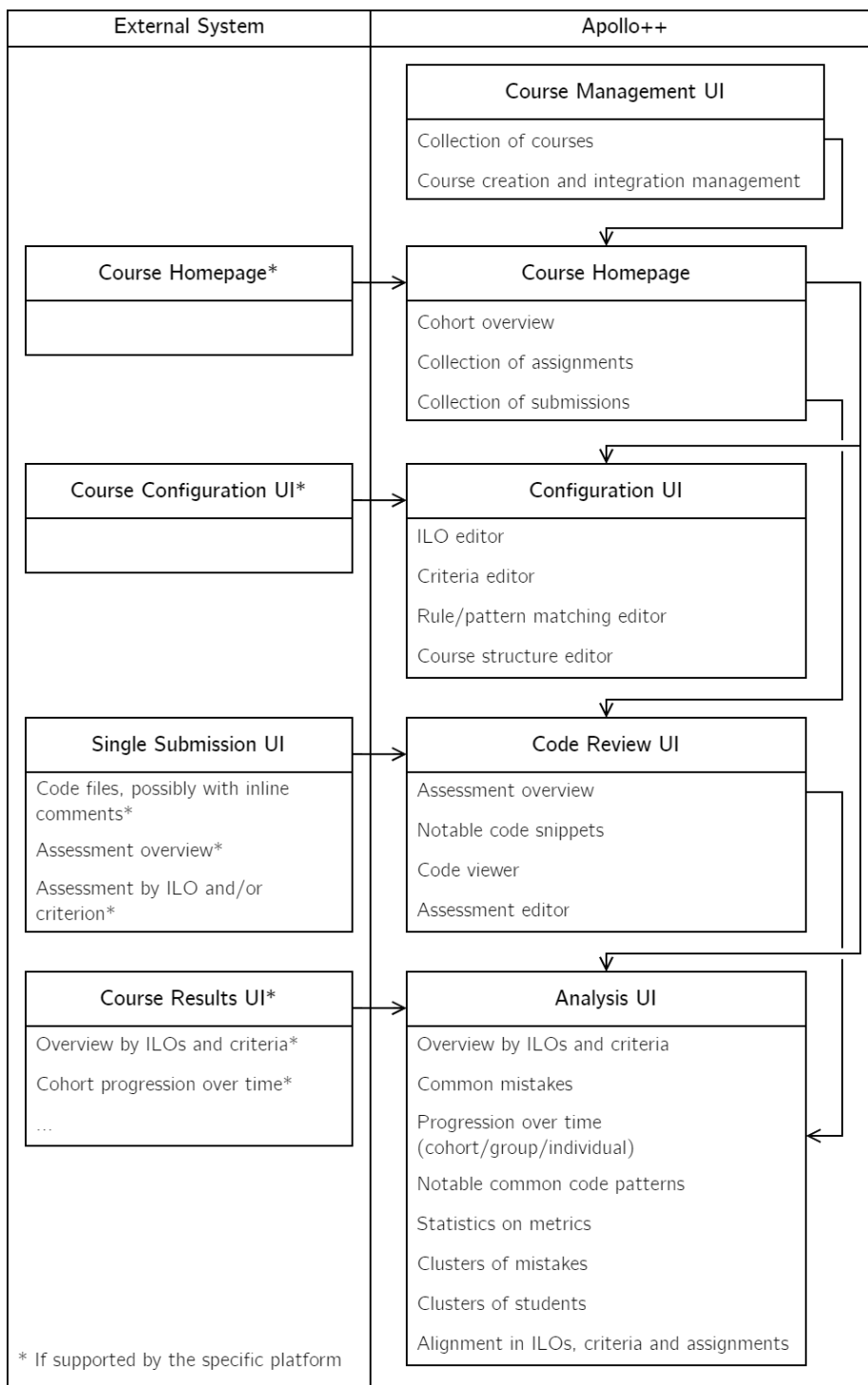


Figure 8.3: Navigation and information diagram for Apollo++, with an external system (like Canvas or GitHub) on the left and the internal Apollo++ UI components on the right.

even with minimal configuration the tool is still useful (Q6).

The *Code Review UI* has a clear overview of the assessment, which highlights some criteria that have clearly been met, criteria for which some counterexamples were found and criteria for which no evidence was found at all (F1, F2, F3, F67). Users can drill down in these criteria to view the relevant code snippets (F26). This part also links to the *Analysis UI* component to see the progression over time of this student (F5).

The main part of the *Code Review UI* shows both the submitted code and all assessment/feedback information. Both parts should include bidirectional navigation to the relevant assessments from some location in the code and to relevant code snippets from some assessment, such that tutors can quickly find the information they are looking for (Q17). The code viewer aids in navigating the project, similar to the tools found in a modern code editor (F23). Terminology in the assessment texts, either automatically generated or provided by a tutor, should link to an explanation of that terminology (F22, F27) and automatically generated texts should use objective language (F24). The author of the feedback (automated or a person) should be clearly shown (F69).

The *Configuration UI* is used to configure the ILOs, criteria, rules and course structure (UC C). Within the Configuration UI, the navigation between those four should be seamless, because while adding criteria, the user may discover a missing ILO, for example (Q13). Ideally, the flow should start with the configuration of ILOs (F32), then the criteria by which those ILOs can be judged and the rules that encode those criteria. Finally, these can be mapped to assignments throughout the course. Splitting these into separate sections helps to keep the UI organised (Q4, Q5), but, in reality, this process is of course not as linear, so the parts should be closely interconnected.

Writing good ILOs is not easy, so the tool should provide teachers with guidance at this point. A community or library where teachers share their ILOs, criteria and rules, combined with some explanation was one of the top requests by configurators (F49), as well as direct support from the tool during the configuration, for example by showing the results on an example project (F54). Configurators also want feedback on which ILOs are assessable automatically by the tool (F55) and which criteria relate to each ILO (F56).

Finally, the *Analysis UI* shows all aggregated information across multiple submissions, to provide feedback to groups of students (UC FC, FG), evaluate the course (UC E, EA), view a students progress over time (UC 2) or perform horizontal grading of projects by criterion (UC 9). The information related to these use cases can be covered in multiple sections or subpages: the criteria that are commonly met or missed (F17, F25); the results of all projects per criterion (F13); progress over time (F5, F14, F40); common code patterns (F18) and statistics on metrics (F19); clusters of mistakes that happen together (F16); clusters of students who make similar mistakes (F15); and the alignment between ILOs, criteria and assignments (F58). For all of these it should be possible to view the data of the cohort, a selected group or a specific student (F5) and to compare different selections (F20). It should also be possible to compare the data with previous editions of a course (F21).

To bring everything together in a single Analysis UI, the main view consists of a

dashboard that shows an overview for each type of information, with the ability to dig in deeper in a detailed view (F57). Given the flexibility required to make comparisons, an existing dashboard and data analysis solution should be used, so that users get full access to all data they find interesting, without the need to integrate all different aspects in the application itself. The tool should come with the common use cases described above already configured.

Overall, the UI makes all functionality of Apollo++ available to users, but we also aim to let users stay within the systems they are already using as much as possible.

Part III

Prototype and evaluation

Based on the design formulated in part II, this part answers our last question: *Is the design feasible and appropriate?* In chapter 9 we introduce a prototype that implements the core parts of the design. In chapter 10, the prototype is used for an empirical evaluation of the tool on real student data, alongside an analytical evaluation of the design. Chapter 11 discusses the results, highlighting its strengths and weaknesses.

Chapter 9

Prototype

We have implemented a partial prototype of Apollo++¹, consisting of the graph-based parts of the *Assessment Pipeline* component and the *Discovery* functionality of the *Assessment Configuration* component. These parts were selected because they form the core of the system and allow us to evaluate the configuration flow and the assessment method with real data. The prototype is built in three parts:

- Extractors for Java and Processing. These take the code as input and output graphs as JSON files. Currently, only the TypeGraph is implemented.
- Graph matcher. This implements the *Pattern Matcher* part of the *Assessment Pipeline* and the *Discovery* functionality of the *Assessment Configuration*, providing a simple textual UI for both.
- LLM suggestions. This program takes in a list of criteria and asks ChatGPT for TypeGraph patterns that could be used to evaluate those criteria.

Each part of the prototype works with JSON files that can be read by the other parts. The extractors output JSON files for the TypeGraph, the graph matcher can read those graphs and patterns defined in JSON and the LLM suggestions are returned in that same JSON format. The following sections describe how each part works in more detail.

9.1 Extractors

The extractor parses and analyses code to extract graph representations used in the other parts of the assessment pipeline. It does so for Java and Processing projects and outputs a TypeGraph (see section 6.1). Processing projects first need to be converted to Java code using the *processing-java* tool included with the Processing IDE. The prototype includes a Python script to perform this conversion for a set of projects exported from the Canvas LMS.

The extractors are implemented in Rascal (Klint et al., 2011), a metaprogramming language for implementing DSLs and performing language analysis. It supports

¹The code of these prototypes is open source and available on GitHub at <https://github.com/art-hurrump/apollopp/>

analysis of Java code based on the compiler from the Eclipse Java Development Tools. We chose to use Rascal for this part, because it comes with comprehensive language analysis tooling built in and builds on an existing compiler, so we can be confident that, for example, name resolution happens correctly.

The extractors are ultimately quite simple, because they rely on the M^3 model provided by Rascal to build the TypeGraph. The M^3 model is a set of relations, mostly between locations. The *uses* relation, for example, relates every usage of a name to its declaration and the *containment* relation links e.g. classes to the methods they contain. The TypeGraph turns this set of relations into a triple relation *from * label * to*, representing an edge-labelled multigraph. The nodes in the TypeGraph are all locations and labels represent the type of relation, like containment, class inheritance, method invocation etc.

The node labels of the TypeGraph are encoded as labelled loops, edges from a node to itself. Encoding node labels as edge labels like this simplifies all code that works with graphs, since it only needs to consider edge labels. This means we can use the type and functions for an edge-labelled multigraph built into Rascal, and we only need to check edge connectivity with the appropriate labels in the subgraph matching algorithm.

The extractors write the TypeGraph for each project to a JSON file, which can then be used by the Graph matcher, as described in the next section.

9.2 Graph matcher

The graph matcher program runs the *Pattern Matcher* in the assessment pipeline to assess projects, as well as the *Discovery* functionality used in the configuration phase. Thus, it has two subcommands: *configure* and *run*. Both take a folder of target projects (as JSON graphs) and a folder of criteria with patterns. Patterns can be defined as a JSON file, or as a JavaScript file that builds an object with the same structure. The latter can be useful to generate many alternative patterns that only vary based on the name of a variable or method. In this section we describe how both subcommands are used and in section 9.2.1 we describe the subgraph matching algorithm in more detail.

The *configure* subcommand watches the criteria folder for changes, and reruns the graph matching algorithm each time one of the patterns changes. At each update, the program prints the results for the given targets, suggestions for new edges between the pattern nodes and new neighbouring nodes that were found in many of the targets or in about half of the targets.

Listing 9.1 shows the output for a pattern that matches every class (to count the number of classes as a metric) on a single Processing project. First, for each project, the results are given: the verdict and which pattern in the pattern tree matched, accompanied by the mapping of pattern nodes to nodes in the target. In this example, the pattern node `class` is matched to the classes `Main` and `Boid` (a subclass of `Main`). Next, the average count of positive, negative and neutral matches over all targets is listed.

Suggestions to extend the pattern are given for each pattern in the pattern tree, in this case only `query_0`. The suggested extensions are all labelled with the

Listing 9.1: Example output for the configure subcommand with a pattern that matches every class, executed on a single Processing project with name 002. The [...] lines show where the output was truncated for brevity.

```

- 002
  Positive: 7
  - Positive / query_0
    - `class` ->
      `java+class:///Main`
  - Positive / query_0
    - `class` ->
      `java+class:///Main/Boid`
  [...]
- Average:
  Positive: 7.0
Suggested extensions:
- query_0
  Edges:
  - [1.00]
    "class"
    > DependsOn >
    "class"
  - [0.29]
    "class"
    > Annotated (NameClass "Particle") >
    "class"
  [...]
  Nodes:
  - [1.00]
    "class"
    > DependsOn >
    | Annotated (ExternalDecl "java+class:///processing/
      core/PVector") |
    = "java+class:///processing/core/PVector"
      in "002"
  [...]
  - [0.57]
    "class"
    > Invokes >
    | Annotated (Modifier "public") |
    | Annotated (InProjectDecl "java+method") |
    = "java+method:///Main/Catapult/display()"
      in "002"

```

fraction of matches that could be extended in that way. Edge extensions indicate the label of the suggested new edge and the two nodes of the query graph it connects. A node extension consists of a node in the query graph, the label of an edge connecting to the new node, the loops that are on that node and which node it corresponds with in some target projects. Since this example was created with only a single project, these suggestions are a little specific (apparently every class in this project makes use of a `PVector`) but when ran on a larger set of projects, this can give more insight.

The `run` subcommand runs all patterns against all targets and outputs a list of results for all patterns for each target project. The results are similar to the results given with the `configure` command: the verdict and identifier for which pattern in the pattern tree, along with the mapping from pattern nodes to target nodes. The output can be written to the command line or to a file for each target.

The next section describes how we implemented the graph matching algorithm to efficiently match TypeGraphs with patterns across many targets and many queries.

9.2.1 Subgraph matching algorithm

The TypeGraph is an edge-labelled loopy directed multigraph, but most off-the-shelf subgraph matching tools only work on graphs with at most one edge between nodes if they even support directed edges and labels. We implemented a variation of the SuMGra algorithm for multigraphs (Ingalalli et al., 2016), but adjusted it to work on edge-labelled loopy directed multigraphs with potentially disconnected query graphs. In this section, we describe the algorithm and the adjustments we made.

9.2.1.1 SuMGra on loopy directed multigraphs

SuMGra works by incrementally growing a mapping, trying to map the next node of the query to all possible candidates from the target. If no candidate is available, the mapping is discarded. To do this efficiently, the algorithm does three things: map the next query node in a specific order, use an index to quickly find initial candidates and use an index to find candidates to extend the mapping. These indices can be calculated once for each target and pattern, so they can be reused when targets and patterns are matched more than once. We will briefly discuss each of these three strategies and how they are adapted to work with directed loopy graphs.

The *query order* is determined by first selecting the most connected node and then selecting the next node based on the maximal connectivity to the already selected nodes. This helps to discard many potential mappings early on, so the search space is pruned early in the process. To support loopy directed graphs, we count both incoming and outgoing edges as connections, while discarding loops.

Initial candidates are selected from a *signature index*. Each node in the target graph is given a signature, a feature vector with features like the number of connecting edges, the number neighbouring nodes (those are not always equal

in a multigraph) or the number of unique edge labels connected to the node. Crucially, the signature of a query node is smaller than the signature of each potential target node: the target node needs at least as many neighbours as the query node, or a mapping would not be possible. These signatures are used as keys in an R-Tree, by turning them into an (n-dimensional) rectangle with a point on the origin and the other at the feature vector. We can then search this R-Tree for all rectangles that contain the point given by the signature of a query node to find all potential target nodes.

To support directed graphs, we changed the features to incorporate the directionality of edges: the number of neighbours with incoming edges, the number of neighbours with outgoing edges, the total number of incoming edges, the total number of outgoing edges, the number of unique edge labels on all incoming and outgoing edges and the maximum number of edges (incoming or outgoing) with any neighbour. We changed the algorithm to support disconnected queries by reusing this index when the next query node is not connected to the previous query nodes.

When a partial mapping is available, next candidates are selected based on the *neighbourhood index*. For each node in the target graph, this index contains a trie keyed by edge labels, storing the neighbours that share edges with those labels. If an edge with labels a , b and c connects node 1 with node 2 , the trie for node 1 has an entry with key $\{a,b,c\}$ storing value 2 . The trie can be queried to find all superset keys, so a query for $\{b,c\}$ would also yield value 2 (and possibly other values). This enables us to find nodes in the target graph that have at least the same edges as the current query node. For efficiency, the trie only stores neighbours and not all nodes. When querying with the empty set, this means that the result will be incomplete, because all nodes have at least zero connections with any node, so really all nodes should be returned.

For this to work with directed graphs, we store two tries per node in the index, one for incoming edges and one for outgoing edges. When searching, we can simply take the intersection of the results for both tries, because we are interested in nodes that include both the requested incoming and outgoing edges. As noted, an empty query should return all nodes, so as an optimization we can avoid taking the intersection and just query one trie if the requested incoming or outgoing edges are the empty set.

9.2.1.2 Efficiently searching with extended patterns

Since the assessment rules are written as a pattern tree (see section 6.2), where each child extends the pattern of its parent, it would be wasteful to search a mapping for each node in the child pattern all over again if we already found a mapping for the pattern in a parent node. Similarly, if no mappings were found for the parent, we can be sure that the extended graph of the child will not have a valid mapping either. To efficiently extend an existing mapping for an extended query graph, we made a few changes in the algorithm: we added another way to order the nodes in the query graph, and we added an entry point to the search that accepts an existing mapping.

Given a query graph Q , we can determine the optimal order for Q according to the rules described before. If we extend this query with additional nodes

and/or edges into query graph Q' to search for extended mappings using the results we already found for Q , we need to ensure that the nodes that are already mapped from Q are also the first in the mapping order for Q' . This can be done by simply copying over the original order and using the rules described before for the additional nodes.

Since the algorithm works by recursively extending an existing mapping, working with an extended query is relatively simple: just pass the original mapping and extended query into the recursive function. There is a catch, however: the extensions to the query may have invalidated the original mapping. If the original query graph contained an edge a from node 1 to node 2 and the extended query graph adds an edge b from node 2 to node 1 , the original mapping may no longer be valid if the target graph does not contain that edge b . For this reason, the entry point that accepts an existing mapping first verifies the validity of that mapping with the given query and target. If the mapping is invalid, an empty list of mappings is returned, because there are no valid mappings that extend the given mapping.

The result is a subgraph matching algorithm that can pre-compute indices for targets and queries, such that these can be reused: the query indices are reused for each target and the target indices are reused with every criterion. The ability to extend the query graph allows us to efficiently work with pattern trees.

9.3 LLM suggestions

The final part of the prototype uses a large language model (LLM) to suggest code patterns that can be used to assess a criterion. The program takes a list of criteria as its input and outputs suggestions for patterns in JSON form. The format is slightly different from what is used in the other parts, listing nodes with node labels and edges separately, because this allows us to more clearly specify which labels are valid node labels and which are valid edge labels. The program is built using the TypeChat² library, which uses TypeScript code to describe the desired output structure to the LLM. We used OpenAI's `gpt-3.5-turbo` model in the prototype, but the tool can be configured to use other supported models through an environment variable.

Listing 9.2 shows an example of a request sent to the LLM. The type definitions are loaded from a TypeScript file, which also includes the structure of the TypeGraph (omitted here, for readability). The results given by the LLM are parsed and checked against this definition using TypeChat, which also runs follow-up queries if the LLM does not respond with a validly typed JSON result. Note that we only request a single pattern with a verdict and not a full pattern tree, in order to keep the structure simple and increase the chances of getting reasonable results. The types also include an explicit failure case, for criteria that can not be assessed in code or using the abstractions provided by the TypeGraph, which should help the LLM to fail quickly and prevent hallucination.

Together, these three parts implement a prototype of the most important functionality in Apollo++: assessment and configuration. In the next chapter, we will evaluate the design using this prototype.

²See <https://microsoft.github.io/TypeChat>

Listing 9.2: Example request to an LLM for the criterion “All fields except constants are private.” The TypeGraph type definitions are omitted here for brevity, but are included in the actual request.

You are a service that translates assessment criteria for programming projects into code patterns encoded as JSON objects of type "PatternResult" according to the following TypeScript definitions:

```
```typescript
export type PatternResult =
 | SuccessResult
 | FailureResult;

export type FailureResult = {
 success: false;
 reason:
 // The criterion is not assessable based on source code
 | "not_code"
 // The criterion is not assessable based on information
 // in the type graph
 | "not_typegraph"
 // Other reason, explained in `message`
 | "other";
 message?: string;
}

export type SuccessResult = {
 success: true;
 patterns: Pattern[];
}

export type Pattern = {
 verdict: "positive" | "negative" | "neutral";
 graph: TypeGraph;
}

// ... TypeGraph definition
```
```

The following is a criterion:

"All fields except constants are private."

The following is the criterion translated into up to 4 patterns as a JSON object with 2 spaces of indentation and no properties with the value undefined:

Chapter 10

Evaluation

In this chapter, we will discuss how we evaluate the design: based on how it addresses the requirements and based on experiments with the prototypes. From the latter, we are interested in the feasibility of the design and the appropriateness of our configuration and assessment methods in practice. In terms of feasibility, we consider the questions (1) if the design can be implemented and (2) whether this results in a tool that could be practically used regarding issues like resource requirements and execution time. For the appropriateness of our method, we consider (3) which criteria can be expressed with the prototype, (4) what support the tool can provide in the configuration process and (5) how the results based on the configured rules match with manual assessments made by tutors.

The methods for the evaluations are described in section 10.1, with the results of the requirement-based analysis in section 10.2 and the results of the experiments with the prototype in section 10.3.

10.1 Methods

The architecture described in this document is evaluated in two ways: based on how the requirements are addressed in the design and using experiments with the prototype, through which we determine the criteria that are supported by the tool and how the results compare to manual assessments.

Our first evaluation method is similar to the scenario-based methods typically used to evaluate software architectures (Dobrica & Niemela, 2002; Patidar & Suman, 2015). In these methods, different scenarios are considered, and the architecture is evaluated on whether those scenarios would be supported. Rather than building new scenarios, we reuse the use cases and requirements that were gathered from the stakeholder interviews, as described in chapter 3. Throughout the document we referenced these use cases and requirements to explain our design decisions, which also gives us a quick way to evaluate which requirements were indeed addressed. We discuss the results of this analysis in section 10.2.

Second, we use the prototype described in the previous chapter to evaluate the feasibility of the design and the appropriateness of the methods, based on the five

questions listed at the start of the chapter. To answer the first question about the feasibility of implementing the design, we rely on our experience building the prototype, which we briefly discuss. For the other questions, we use the assessment criteria for the Software Systems (SS) and Algorithms in Creative Technology (AiC) courses introduced in chapter 3 and programs submitted by students in previous editions of these courses, as well as the assessment forms for those programs. We received this data from the respective teachers of each course, asking them for permission to use their assessments and report on it in this report. We provided the teachers with a script to anonymize the code and assessments before sharing it with us, so that we would not receive any personally identifiable information of the students. We did not ask students for their consent, because we are dealing with anonymous data, the students do not personally participate, nor are they the subject of this research, and we do not publish or share their code (to which students retain their copyright).¹

The list of criteria for the evaluation is based on the course materials. For both courses, the project description includes a list of requirements that each project should meet. In SS, this includes some detailed functional requirements, like²

- When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.

and for AiC, the project description specifies, for example, how much original work should be in a student's project.³

- Flocking and particles should not both be from Shiffman (or other sources on the web), at least one of them self written.
- It will count how much code you have written yourself. A major part of significant complexity has to be written by yourself.

Both projects also include a rubric, which describes the requirements in more detail and categorized by subject. For SS, this is a full rubric with different criteria per level of achievement. In AiC, the cells of the rubric are left open, but they indicate different levels of achievement of the criteria listed per block. From these project descriptions and rubrics, we extracted a flat list of criteria to use for the evaluation. Note that this means that a perfect submission does not fulfil all requirements, since some of them are from lower achievement levels in a rubric. The full list can be found in appendix A.

The student programs we used were stripped of all files except for code files and included dependencies, in order to prevent deanonymisation through, for example, images. Within the code files, we replaced all occurrences of student names and other identifiers with randomly generated identifiers, such that comments and file names were also anonymized.⁴ The assessment forms we received consist of a few comments per category in the rubric, which provide justification for

¹The usage of anonymized student data for the outlined evaluation was approved by the Ethics Committee Computer & Information Science of the University of Twente. This experiment is known to them as application *230346*.

²From the Software Systems manual dated 18th January 2022, page 22.

³From the reader for Algorithms in Creative Technology used in the 2021 edition, page 23.

⁴The script that performs the anonymization is available at <https://github.com/arthurhump/apollopp/>, together with the prototype code.

a certain level of achievement for that category. For SS, we used data from the 22/23 edition with 170 submissions, of which 3 were unusable because the students packed their source files in a *.rar* archive, which was not supported by the anonymization script. For AiC, we used data from the 20/21 and 22/23 editions, with 53 and 39 submissions respectively.

To answer questions (3) and (4) about the supported criteria and configuration, we considered all criteria from both courses and tried to configure rules in the tool for each. The result is a pattern if successful or an explanation why this criterion does not fit in the tool if not. Based on the full process, we could also evaluate how well the tool supported us in the configuration process. Here, we considered the usefulness of suggestions generated through an LLM and the suggestions provided by extending a pattern with nodes and edges found in student programs. We used a random subset of 20 programs from each course to serve as test programs during the configuration phase. This means that suggestions for new nodes and edges were extracted from these 20 programs.

Finally, we used the configured rules and the submissions to answer question (5), by running the tool and comparing the results with the assessments by the teacher. Here, we ran the tool on all submissions, except for those 20 that were used in the configuration phase, and compared the results for 10 randomly selected submissions in each course. The tool reports the amount of positive and negative matches found for each criterion, as well as specific locations, which we compared with the notes and scores given in the rubric by the teacher. The main classifications are *Agree* if the manual assessment is in agreement with the results of Apollo++ (e.g. all fields are indeed private), *Disagree* if that is clearly not the case, or *Rubric-not-mentioned* if that criterion is not mentioned in the filled-in rubrics. Note that there are different reasons why a criterion may not be mentioned, for example if a project does not meet the criteria set out for the “sufficient” level, there is no point to mention the criteria on the “good” level. On the other hand, it might also be an oversight by the assessor. Besides those three categories, some criteria require more information. In AiC, some criteria mention “how much” there is of something, like “How many classes are there?” (12). In those cases we recorded the number reported by Apollo++, as well as what is mentioned in the rubric, because the criteria are not clear on “how many” is considered sufficient or good. Some other labels were added to include relevant context, which will be explained in the results.

The results related to all 5 questions are described in section 10.3.

10.2 Addressing requirements

First, we review how the design addresses the requirements that were gathered from the stakeholder interviews. Table 3.1 shows for each requirement which chapters address it in the last column. Out of 89, we have not addressed ten requirements, four of which were already marked out of scope. The other six were all mentioned just once during the interviews:

- F29. Can separate parts of criteria that change (e.g. rules of the game) and parts that stay the same (e.g. overall structure)
- F68. Is less hesitant in the wording than to tutors

- F70. Shows assessment as a rubric
- Q7. Rubrics are easy to import and export
- Q11. Works correctly and does not spew error messages
- Q12. Provides option to tweak every little setting

We decided not to include provisions for F29 in the datamodel, since the sharing model works per criterion and thus there is no need to differentiate between these in the system. If a configurator wants to use this distinction to structure their criteria, they can simply use the categories that are present in the model. From the project descriptions for SS, where this issue is relevant, we see that this is already how the criteria are structured, with the game logic criteria relegated to their own category. Likewise, we have not directly addressed F68, but tutors are able to extend and/or modify the feedback before it is presented to students (addressing F8), so they could change the wording if that is desired. In the prototype, no texts are presented beyond the literal text of the criterion and whether an example is positive or negative, so there is not much to adjust.

F70 and Q7 are both about rubrics, for which we have not made any specific provisions. The reason is that we focus on providing feedback, rather than on calculating grades (following F12, which was mentioned by six participants). Thus, it is more important to get feedback on each specific criterion, rather than seeing what score you would receive according to a rubric. Importing rubrics would also be a difficult endeavour, since there is no standard format for rubrics and a platform like the Canvas LMS does not support exporting rubrics created on their platform.

Q11 is possibly the biggest omission from our design, for which there indeed is little consideration. We do not believe there are design decisions that directly oppose this requirement, but the many integration points might make failures more likely. We have not made any statements on development techniques and practices, which could help to address this requirement.

Finally, we have not addressed Q12 in this design. We have not specifically addressed tweaking small details, mainly because many of those small details are not specified in the design. The workflow in general, however, provides a very flexible assessment model which certainly does not prohibit this configurability when it comes to the assessment.

Overall, this shows that the design is successful in addressing the requirements, even considering those that were not explicitly addressed.

10.3 Prototype experiments

In section 10.1 we listed five questions that can be answered using the prototype. The first two questions about the feasibility of the design are answered in section 10.3.1 and the last three about the appropriateness of the assessment method in section 10.3.2.

10.3.1 Feasibility of the design

First, we asked if the design can be implemented in practice. Given the description of the prototype in chapter 9, the answer to this question is clearly *yes*, at least

for those parts that were implemented. Since those form the core functionality of the system, we can be confident that the system as outlined is feasible to implement.

Second, we asked whether the tool resulting from this design would be practically usable regarding resource requirements and execution time. To answer the remaining questions, we ran the tool on real data from two courses, for which we also took some time measurements to answer this question. In terms of resources, we used a recent laptop with an AMD Ryzen 9 5900HX CPU and 32 GB RAM running Windows 11. First, we ran the *Extractor* to generate the TypeGraphs from the source projects. On the 167 projects for SS, this took 274 seconds in total, or about 1.6 seconds on average per project. On the 92 projects for AiC, this took 18 seconds in total, or about 0.2 seconds on average per project. This difference is likely due to the difference in size between the projects for those courses, and the fact that the Processing compiler writes its Java output to a single file and the extractor thus only has to read one file per project.

During the configuration phase, we ran the patterns for one criterion on 20 submissions at a time, while also searching for possible extensions. Depending on the complexity of the pattern and the size of the projects, this took a few hundred milliseconds for most criteria, and about five seconds for two of the more complex criteria (AiC 11 and SS 63, see appendix A).

After configuration, we ran the tool with all patterns on all remaining submissions. For SS, we ran 10 criteria against 147 projects (with an average size of about 24 classes). This took about 26 seconds total, or less than 200 milliseconds per project on average. Of the 26 seconds, 16 seconds were spent on parsing the target project TypeGraphs and building the graph matching indices, 100 milliseconds on parsing the criteria and preparing those for matching and 10 seconds on actually running the graph matching algorithm and writing out the results. For AiC, we ran 9 criteria against 72 projects (with an average size of about 12 classes). This took slightly above 7 seconds, or less than 100 milliseconds per project on average. Parsing the TypeGraphs and preparing the indices took about 2 seconds total, reading the criteria again about 100 milliseconds and running the graph matching about 5 seconds.

Overall, this shows that the approach is certainly feasible for providing results on demand, like when giving feedback during a tutorial session.

10.3.2 Appropriateness of the assessment method


Next, we are interested in the appropriateness of the assessment method we implemented: can it perform the assessments we want it to perform? This means that the criteria of a course should be configurable in the tool, and that the results given by the tool should align with the assessment made by a human tutor. Given the effort involved in the configuration, the tool should support that process to make it as easy as possible.


10.3.2.1 Configurability of criteria


The third question we asked is which criteria can be expressed with the prototype? We went through the list of criteria for the SS and AiC courses, trying to configure


patterns in our prototype for each, assisted by the LLM-generated suggestions and the proposed extensions derived from the set of 20 projects we used to test the configuration while working on it.

Appendix A lists all criteria and whether we were able to configure a rule within the prototype for that criteria. For SS, we were able to configure 9 patterns in the prototype, out of 81 criteria. Since we only implemented the TypeGraph in the prototype, all patterns use that program representation. These patterns do not all fully cover the criterion, but match an important part of it. For example, criterion 78 from SS states “Custom exceptions are defined where appropriate and no equivalent predefined exceptions exist.” We configured a pattern to match custom exceptions, but were unable to devise a pattern that separates the appropriate from the inappropriate custom exceptions.

Figure 10.1 (the right side for SS) shows the results as a Venn-diagram, with criteria which were partially implemented as the intersection of two results. The green region marked with  indicate the criteria for which a pattern was configured. On the left, the results for AiC are shown, where we were able to configure patterns for 10 out of 27 criteria.

The blue region marked with  in both diagrams are the criteria for which we think it would be possible to define a pattern using a PDG rather than the TypeGraph. There are 11 of these in SS and 8 in AiC. An example from SS is “When the client is started, it should ask the user for the IP-address and port number of the server to connect to.” The TypeGraph does not have sufficient information to find this, but with a PDG we could look for data flowing from console input to the method that is called to build a socket connection.

For 40 criteria in SS and 10 in AiC, we were unable to define a pattern based on the TypeGraph, but it may be possible to find patterns using other program representations and still configure a pattern within the system. This is the purple area in the diagram, marked with . An example is criterion 24 from AiC “Have comments and a header.” This is not assessable through the TypeGraph (which does not include comments), but it is totally feasible using a different program representation. For others in this category, it is less clear that a different program representation would be sufficient, like many of the functional requirements for SS.

Finally, 26 criteria from SS and 6 from AiC were marked as incompatible with our framework, with static analysis in general or just difficult to assess automatically, because they rely on a deeper level of understanding. This is the red region, marked with . In AiC these are criteria like “Functional correctness” (25) and “Attention for details” (27), which are quite broad and thus difficult to assess automatically. In SS there are some large functional requirements which would be unfeasible to assess with static analysis or criteria like “There are no compilation errors or failed tests” (44) or “No Checkstyle violations” (49) which rely on running tests or an external tool, which is not clearly viable in our design.

The last type of criteria in this category, are those which relate to “clearly defined components” (67) and separation between those components like “The UI is separated through interfaces such that other components do not know the concrete UI classes” (74) or “The game logic is not aware of any concrete classes of the rest of the program.” (75) With the current pattern-based assessment

pipeline, there is no clear way in which those components could be identified and used in patterns.

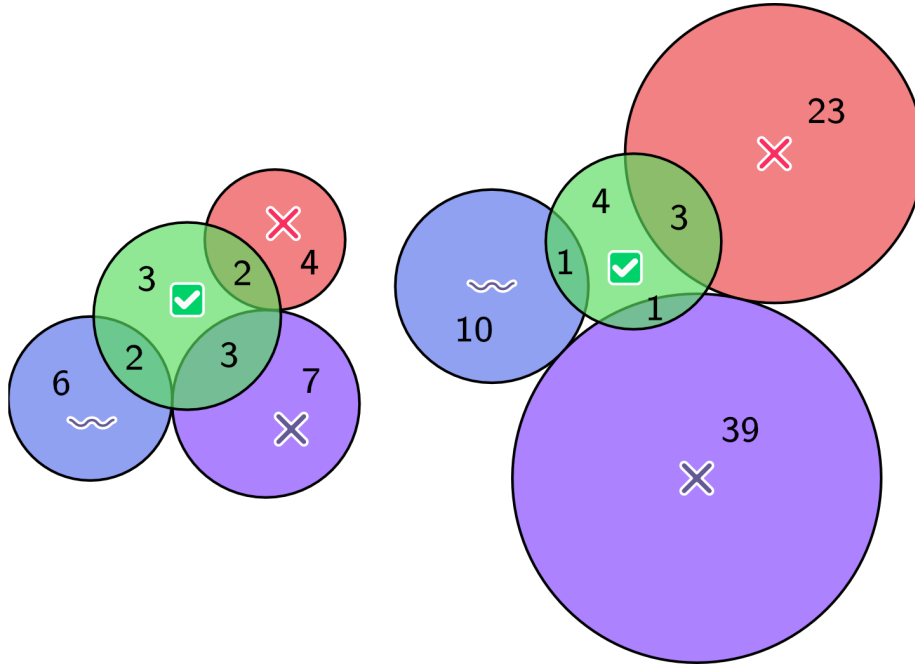


Figure 10.1: Overview of the configuration results, showing the number of criteria that could be configured (✓), that could be configured with a PDG (wavy line), that can not be configured using the TypeGraph (X) and that is not configurable in our design or not at all (X). The bubbles on the left show the results for AiC, those on the right for SS.

Overall, we see that we were able to configure patterns for a larger portion of the criteria for AiC than for SS. This is the case because the criteria for the former are more focussed on the design of class relations and the common functionality has a clear class structure, for which the TypeGraph is particularly well suited. For the more detailed functional requirements in the SS project, a different type of program representation is needed, which could make them work within our design. Some criteria do not fit within the proposed structure, while others are simply difficult to assess automatically in any way.

10.3.2.2 Supporting the configuration process

The fourth evaluation question asked whether the tool is able to give adequate support during the configuration phase. We implemented two configuration functions in the prototype to investigate this: suggestions for patterns based on the criterion text, using an LLM; and suggestions to extend a pattern discovered from a set of previously submitted projects.

Starting with the LLM-provided suggestions: these were almost always useless. As described in section 9.3, we allowed the LLM to respond with an error if a criterion was not automatically assessable or not assessable using the TypeGraph,

but this response was not used for any of the criteria. Given the results in the previous section, that is not what would be expected. Instead, it returned patterns that used `nontrivial` as a modifier on a class to detect “at least two (nontrivial) classes written by yourself” (AiC 6), even though we specified a list of supported modifiers.

All nonsensical responses aside, even for the criteria that are assessable using the TypeGraph, the responses were not really helpful. For the (simple) criterion “All fields except constants are private,” (SS 62) the result contained two suggested patterns: one negative for fields with a `public` modifier and one positive for fields with a `private` modifier, thus totally ignoring the “except constants” part of the criterion.

Suggested extensions to patterns were more helpful, because they are actually found in real programs, so at least they have a basis in reality. However, there are many possible ways in which to extend a pattern such that one or more previous occurrences would match, and only a few might actually be relevant. In our prototype we highlighted those suggestions for which about half of the original occurrences would match the extended pattern and half would not, but that still results in many unhelpful suggestions. Because the suggestions are restricted to only a single neighbouring node, more complex extensions that would be helpful can not be found.

Overall, our prototype was unable to provide good support in the configuration phase.

10.3.2.3 Making reasonable assessments

Finally, we asked how the results returned by our tool compare to the assessments made by humans. We compared the results reported by the prototype for 10 projects of each course, the results of which are listed in appendix B. Here, we will briefly discuss the results for each of the criteria, starting with those from AiC, followed by SS.

Starting with 2 “Includes meaningful randomness (normal distribution and Perlin noise),” we find that the tool and the assessment agree on 8 out of 10 submissions and disagree on the other 2. In one disagreement, the tool did not find a usage of Perlin noise which was mentioned in the rubrics, while in the other the tool did find one use of randomness, but it was not mentioned as one of the implemented topics in the rubric. The latter may still be correct is that usage was not “meaningful”.

For 4 “Includes particles,” we find clear agreement in 4 cases, clear disagreement in another 4 and in 2 cases the criterion was not mentioned in the rubric. In all cases with disagreement, our tool did not find the particle system that was present according to the rubric. Upon further investigation, in those cases the students used an array rather than an `ArrayList`, which is not easily representable in the TypeGraph, which is likely why that was missed in the configuration phase. For 5 “Includes flocking,” which is similar to the particle system, we find agreement in 7 cases, disagreement in 1 case and 2 cases where there is no mention in the rubric either way.

For 11 “How much interaction is between classes,” and 12 “How many classes are

there,” we recorded the numbers reported by the tool and either *Agree/Disagree* for those cases where especially high or low numbers were called out in the rubric or *Rubric-positive/negative/not-mentioned* for all other cases. For the interaction criterion we find 4 outliers where the tool reports a high number, but the rubric is negative. These are programs where students created a God-class as a replacement for the main tab, so now every interaction in the program is seen as interaction between classes. In the other cases we see that the highest number is marked positively in the rubric and the lowest negatively, so there seems to be some agreement here. The number of classes is only mentioned in the rubric in one case, which is a program with 17 classes (on an average of 12).

“Do not use unnecessary global variables, especially not for value passing” (14) is mentioned twice in the rubrics, both cases where the tool also reports the usage of a global variable in multiple classes. For one of these, with 23 reports, 20 are about the same variable, because that variable was used in many classes and thus the pattern matched many times. In 3 cases, no global variables were reported, because the program used a God class and the main tab (where global variables are defined) was basically empty. All other 5 cases where the rubric does not mention the global variables, the tool did find some. The pattern does not distinguish between only accessing a global variable and treating it as a constant or changing a global variable, so these might indeed not be problematic global variables.

For 17 “Do not hide user interaction in classes,” we find 5 agreements and 2 cases where the rubric does not mention this criterion. In the 3 cases with disagreement, we found that the students did use the event variables outside the event handling methods, but only within methods that were called by those event handling methods, which is considered fine by the assessor, but was not expressed in the pattern. While we could add neutral patterns for methods that are called by event handling methods directly, there is no way to express that it should appear somewhere in the callstack from such a method if there are “intermediate” methods.

Finally, for AiC, the criteria 21 “Use functions when you have similar code,” and 23 “Do not have unused code” were not mentioned in any rubric, so there is nothing more we can report on those.

For SS, criterion 50 “Constants are used where appropriate,” is mentioned in 5 cases, all of which are in agreement with the tool, which reported a large number of constants. In the other 5 cases, the rubric makes no mention of this criterion. For 59 “The server is tested with unit tests that check whether the protocol is handled correctly,” we find agreement in 9 cases. For 1 case there is no mention of this criterion in the rubric.

With 61 “Classes do not access the state of other classes directly,” we find agreement in 8 cases and disagreement in the other 2. One of these included many “false positives”, because their test cases accessed the state of some classes, which is considered fine for the overall structure of the system. Relatedly, for 62 “All fields except constants are private,” we find agreement in 6 cases and disagreement in 4. For at least 2, it seems the assessor overlooked a few fields or decided not to judge too harshly on one or two fields that were not private.

Criterion 63 “Methods are public only if they are intended to be used by other

classes” fared less well, with disagreement in 6 cases, agreement in just 1 and 3 cases where this was not mentioned in the rubric. Here the issue of intention came up, because the tool flagged various methods that were not called by other classes, like a few implementations of `toString` which were not marked with `@Override` and thus not detected as an override. These are clearly intended to be used by other classes, and may have been in some debugging code, but were not in the final project.

We found agreement in all 10 cases for 66 “There is more than one component (package).” Also, all projects did have more than one package.

“Interfaces are applied where appropriate to ensure low coupling between components” (69) is not mentioned in 4 cases. For the other 6, we found agreement for 1 and disagreement for the other 5. In one of these cases, where the tool reported not finding any interfaces, the rubric still gave full marks for structure (of which this is one of the criteria). Upon closer inspection, this project did indeed have a nice and clear structure with clearly identifiable components and classes, so this may have been a conscious decision to deviate slightly from the literal criteria.

For 77 “A creational pattern such as the Factory pattern or dependency injection is used appropriately,” we only implemented a pattern for the dependency injection part with the caveat that this would be more precise when expressed using a PDG rather than the TypeGraph. Except for 1 case where the criterion is not mentioned, we indeed found disagreement for the other 9 cases. For 7 of these, our pattern resulted in many false positives, reporting simple property initialization, rather than dependency injection.

Finally, for 78 “Custom exceptions are defined where appropriate and no equivalent predefined exceptions exist”, we found agreement for 9 cases where custom exceptions were or were not created and the rubrics made the same assessment. In 1 case, the rubric made no mention of this requirement.

Figure 10.2 shows a summary of these results, with the number of projects where we find agreement, disagreement, or no mention in the manual assessment per criterion. In the AiC criteria, we clearly see that the results always come with a caveat, whether a correct implementation is missed or the criterion is more nuanced than what can be detected using patterns. On the SS side, we see the patterns performing either quite well or quite badly. The latter is especially the case where terms like “appropriate” are used and the best our tool can do is flag parts of the code that may be of interest when making a decision.

This concludes the results that answer our evaluation questions on feasibility and appropriateness. In the next chapter, we will discuss the results and the strengths and weaknesses they reveal in the design and suggest some ways in which the design may be improved.

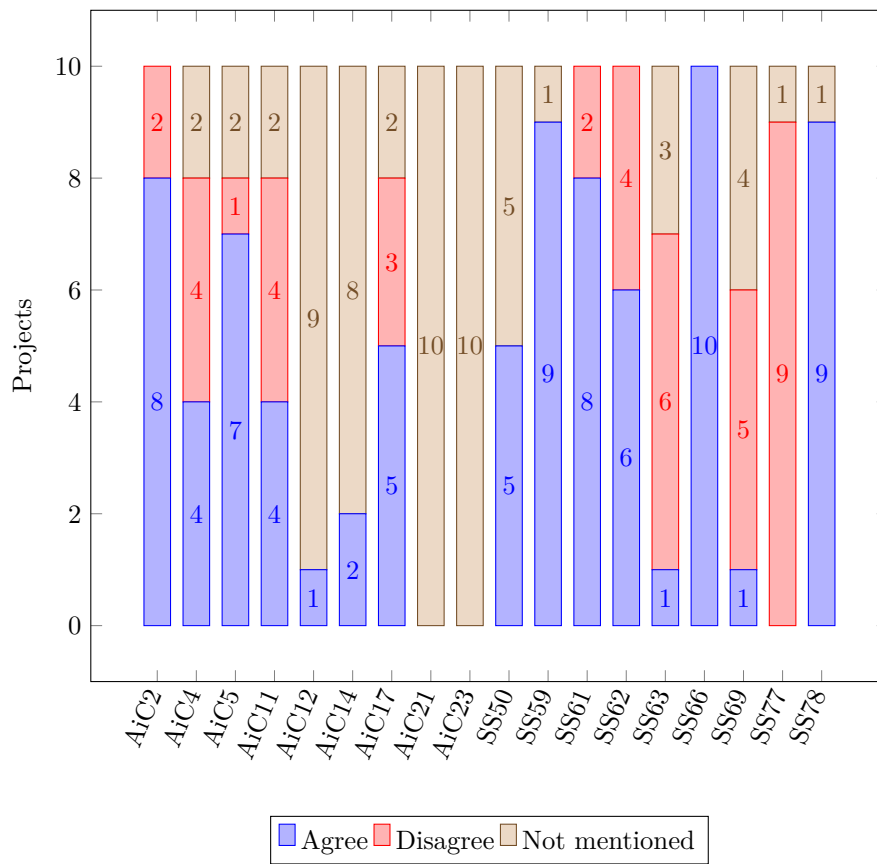


Figure 10.2: Agreement between tool results and manual assessment, for each of the configured criteria.

Chapter 11

Discussion

In this chapter, we discuss the strengths and weaknesses of the proposed design for Apollo++ based on the results from the previous chapter. We also suggest ways in which those weaknesses may be addressed.

11.1 What works well

To start positively: the feasibility of the design seems good. We were able to implement a prototype without too many issues and in terms of performance and resource requirements, subgraph matching is fast enough to provide feedback in live situations. For the actual assessment process, abstracting the code into different graph representations and working with these graphs looks like a good approach to the problem.

As foreseen in the design and confirmed through the experiments, implementing different criteria requires more expressiveness than is available in a single program representation (the TypeGraph, in our experiments). The design foresees the use of multiple types of graphs, so this fits with the design. The different program representations could also be used to do more complex pre-processing, like identifying the components in a project, even though this feature was not initially intended for that use case.

11.2 Patterns need more expressiveness

A weakness to this point is that simple patterns are quite restrictive: in the prototype we added the functionality to generate patterns using JavaScript code, rather than just writing out the patterns, because we wanted to express a few alternatives for one node in patterns that are otherwise the same. An example is the AiC criterion that checks for usage of one of the random number generators available in Processing. All patterns for this criterion are the same, only varied by the actual method called. This suggests that a UI that only allows the user to define patterns as graphs would not be sufficiently powerful, especially to users who are familiar with programming and the abstractions it can provide. This can be improved in two ways: by extending the pattern language to support such

abstractions (extending the DSL with general-purpose language (GPL) features), or by defining patterns in a GPL that already supports abstractions. In our prototype we chose the latter option, because the implementation took only three lines of code. Generating the pattern in a GPL does mean that information about the structure is lost to the tool itself, which makes suggestions to extend the pattern (or do so from other parts of the UI) more difficult.

Another issue that we foresaw in the design is that fully automated assessment is not feasible: automation can only do so much and humans are better able to make a nuanced assessments and judge notions like “meaningful” and “appropriate”. This is why we explicitly allow tutors to extend and change the feedback, before showing it to students. One thing missing from the design, however, is a way to make this distinction between automatable and partly automatable when configuring rules. Some criteria are objective and allow little variation, so rules for those can be seen as providing a real assessment. On the other hand, for a criterion that checks if “custom exceptions are defined *where appropriate*” (SS 78), we can find those custom exceptions, but checking appropriateness is much more difficult to automate. In the current design, there is no way to distinguish such rules.

Related to the properties of criteria and patterns, for some patterns we found that a pattern matches many times, while we are really only interested in one node that all matches have in common. An extreme example is the criterion that “there is more than one component (package)” (SS 66). The pattern we configured matched every pair of packages, but a more reasonable and useful result would be a single match on the set of all packages. Similarly, the pattern that matches tests that test the server code (SS 59) matches a test for each method of the server called by this test, while we would rather only have the test method returned once if there are one or more calls to server methods. Lastly, for global variables, while it would still be interesting to also see where they are used, it would make sense to group those matches by variable. Overall, patterns would need a way to select “nodes of interest” to support these criteria better.

11.3 The pipeline could be more flexible

Some other criteria were difficult to configure in the tool because they did not quite fit within the assessment pipeline. For some, the pipeline itself is a problem: one of the criteria for AiC is that a project “combines at least 3 from the 4 topics” (1), where each of the topics is covered by its own criterion. Assessing this criterion thus depends on the results of other criteria, which is not possible within the pipeline model.

Another issue with the pipeline as proposed, is that there is no way to incorporate other tools. In SS, several criteria are about the project compiling without errors, all tests passing and there being no Checkstyle violations. To assess those criteria, external tools like a compiler, test runner and Checkstyle are required. This could be solved by allowing “assessors” to work at any level of abstraction: running directly on the files on disk, on the text of the code, on the AST or one some other graph representation. The pattern matcher would just be one assessor in this system, with others being able to run a compiler or Checkstyle.

In the current design we accounted for patterns and metrics as the two “built-in assessors”, but these criteria show that this principle should be generalized.

While we considered the need for humans to be part of the assessment process, the design only allows information to flow from tool to human and not the other way around. In SS, many requirements reference the components of a project, which can be tricky to identify because students rarely create perfectly separated components (or they all would score perfectly on structure). In this case, it may be helpful to have a tutor identify the main components and use this information to assess some of the criteria that depend on it. One way to do this is to split assessment into “identification” and “quality assessment”, where the first stage identifies parts of the program to be assessed. Then those parts are assessed on their quality. In some cases the first stage is automatable (e.g. finding all custom exceptions) and judging the quality is difficult to automate (are those exceptions appropriate?). In other cases, like the dependencies between components, identifying the components is more difficult to automate, but once they are known, it is simple to find the dependencies. Whether this structure could apply generally, or whether this approach should be generalized to support more situations, is not clear at this point.

These last three issues we mentioned require significant changes to the assessment pipeline. Clearly, the current design is too simple to support these scenarios, so a more complex solution is needed. The main challenge here is to keep this component just simple enough to support these criteria, but still remain understandable (because the assessment process should be transparent, in the sense that students should be able to understand how an assessment is formed).

11.4 Configuration needs better support

Finally, the configuration support in our prototype is not great. The LLM suggestions are practically useless. While the performance could likely be improved by including some examples of criteria and patterns in the prompt, we suspect that the level of abstraction of a graph representation of code is too high for an LLM to generate useful results, because its training data most likely did not include many graph representations of code. We note that this may not only be a problem for artificial intelligence, but also for human intelligence. For both, it may be helpful to write patterns in a way that is more closely related to the code they try to represent, rather than as a list of labelled nodes and edges.

The other supporting feature we added, the suggestions for extensions of a pattern with nodes and edges found in other projects, did not perform well either. There are many suggestions for each pattern, of which only a small number are actually useful. Given our previous observation about the level of abstraction of these patterns, it may be more helpful to let the configurator select concrete examples from example projects and past submissions and distilling patterns from those limited examples. This could be combined with the “Refine graph patterns based on encountered examples” functionality in the Assessment Configuration, to turn the configuration into a more continuous process as more information (in the form of new projects) comes in.

Overall, we see that the basic approach of our design works: it is certainly

feasible and appropriate for at least some criteria. There are three major areas of improvement, however: the way patterns are described needs to be more expressive, to better support complex patterns and the nuances of human-machine collaboration in the assessment process; the pipeline can be more flexible, to support more types of criteria than can be done with simple patterns and metrics; and the configuration phase needs better support, with a more intuitive way to write patterns and a better way to make use of examples and past submissions.

Part IV

Conclusion and backmatter

Chapter 12

Conclusion

In the introduction we asked *how, and to what extent, can an automated assessment tool assess larger programming projects?* To answer this question, we asked four more questions to cover the different aspects in our main question. In this chapter we review the answers we found to those four questions and then formulate an answer to the main question.

What does assessment of projects look like, both manual and automated? In chapter 2 we introduced intended learning outcomes (ILOs), which describe what a student is expected to learn, and assessment criteria, which describe the expectations for the artefact that is to be assessed in more detail. We also discussed the *ill-definedness* of projects, which means that there is no definite solution against which a student's solution for a project can be checked. Finally, we reviewed the state of the art for automated assessment tools and found that they are mostly built for small, well-defined exercises. Tools that do aim to work on more ill-defined exercises typically rely more on student data and focus on the criteria that are well-defined.

What are the requirements for an automated assessment tool that assesses projects? As described in chapter 3, our tool aims to automate part of the assessment process for large programming projects where students have a lot of freedom to choose how they implement their solution. From interviews with stakeholders, we determined that this tool should be *flexible* and *supporting*, meaning that it supports as many assessment criteria as could be automatically assessed, while also integrating in different environments and platforms. It needs to support teachers in the assessment process, rather than performing fully automated assessment, and also support them during the configuration process to get the tool up and running.

How to build this tool? In part II we introduced a design for Apollo++ that fits those requirements. The core of this design is an assessment pipeline that turns code into different graph representations, against which graph patterns are matched to build an assessment. These graph patterns are defined based on the assessment criteria, which are already in use by the tutors who assess these projects manually. The design explicitly includes teachers in the feedback loop, which addresses one of the major concerns voiced by our stakeholders.

Is the design feasible and appropriate? In other words, does it run and does it give reasonable results? We implemented some parts of our architecture in a prototype, to see how it works and performs in practice. This showed that the design is feasible to implement and is able to perform assessments fast enough to be usable in an environment where direct feedback is desired, like in a tutorial. We were able to configure patterns for some criteria in our prototype and expect many more would be feasible with more graph representations. The results of running those patterns on real projects shows that we find reasonable results for most patterns, but, as expected, human review is still necessary for many.

When it comes to *flexibility* and *support*, however, the tool is not yet at the level where we want it to be. There are still some criteria that do not fit in the model that our design proposes, even though they are (partly) automatically assessable in principle, so the model is not quite flexible enough. The features that should support teachers in the configuration phase were not helpful in practice. While the LLM-suggested patterns were good for a laugh, they did not help to actually configure any real patterns. The configuration process is still a major area where improvements can be made, to actually support teachers in using the tool.

So, to come back to our main question: *how, and to what extent, can an automated assessment tool assess larger projects?* We have shown that our design works for some criteria and can likely support more within the same structure. Thus, starting from criteria, using multiple program representations and defining patterns for subgraph matching certainly is a way how an automated assessment tool can assess larger projects.

To what extent depends on the criteria, the available program representations and the pattern configuration. With our prototype, the extent is rather small: only 11% (in SS) to 37% (in AiC) of the criteria could be (partially) configured and for some of these the disagreement with manual assessments was quite large. This is partially due to the loss of some nuance in the criteria, where the reported results would still be useful for an assessor to inform their judgement. With more program representations and better pattern configurations, however, we believe that these results can be significantly improved. Still, given the ill-definedness of some criteria, it is unlikely that assessment of these projects can be fully automated.

Acronyms

| | |
|---------------|-----------------------------------|
| AGS | Assignment and Grade Services |
| AiC | Algorithms in Creative Technology |
| AST | Abstract syntax tree |
| CreaTe | Creative Technology |
| DSL | Domain specific language |
| GPL | General-purpose language |
| ILO | Intended learning outcome |
| ITS | Intelligent tutoring system |
| LLM | Large language model |
| LMS | Learning management system |
| LTI | Learning Tools Interoperability |
| PDG | Program dependence graph |
| SS | Software Systems |
| TCS | Technical Computer Science |

Glossary

Abstract syntax tree (AST) Abstract representation of code as a (tree-shaped) graph. An AST is produced by parsing the textual representation of code and typically does not contain irrelevant details such as the amount of whitespace (for languages where that is irrelevant).

Actual learning outcome What the student has actually learned at the end of a study unit. This can be compared to the intended learning outcome.

Algorithms in Creative Technology (AiC) A course about programming in Processing for the fourth module of CreaTe. This course assumes some prior programming experience and has a focus on programming style and good structure.

Assignment and Grade Services (AGS) An LTI specification around assignments and grades, which can give tools access to the results for an assignment and allow them to publish scores back to the platform.

Atelier Platform for feedback during programming tutorials, with support for automated feedback tools. See (Fehnker et al., 2021) for more details.

Automated assessment Automated assessment tools automatically assess, score or grade artefacts produced by students.

Creative Technology (CreaTe) Multidisciplinary programme at the University of Twente with a base in computer science and electrical engineering and a focus on design.

Domain specific language (DSL) A programming language specialised to a certain domain. These languages are often less powerful than general purpose programming languages, but they typically are also simpler to use for non-programmer domain experts.

General-purpose language (GPL) A general-purpose programming language can be used to build applications across many domains, as opposed to a DSL.

Intelligent tutoring system (ITS) Intelligent tutoring systems provide feedback to students, often interactively while a student is working on an exercise.

Intended learning outcome (ILO) Statement of what a student is expected to have learned at the end of a study unit. This describes the *intention*

of what a student will learn, which is not necessarily their actual learning outcome.

Large language model (LLM) A neural network trained on a large repository of texts to generate texts based on a starting prompt.

Learning management system (LMS) A system on which course materials are shared and where students can hand in assignments. Popular LMSs include Blackboard, Canvas and Moodle.

Learning Tools Interoperability (LTI) A set of specifications that enable integration between learning tools implemented by many LMSs.

Program dependence graph (PDG) A graph representation of code which combines control dependencies and data dependencies in a single graph.

Software Systems (SS) Second module of TCS. This module has a focus on programming and software design.

Technical Computer Science (TCS) Bachelor programme in Computer Science at the University of Twente.

References

- Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2), 83–102. <https://doi.org/10.1080/08993400500150747>
- Balse, R., Valaboju, B., Singhal, S., Warriem, J. M., & Prasad, P. (2023, June). Investigating the potential of GPT-3 in providing feedback for programming assessments. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education v. 1*. <https://doi.org/10.1145/3587102.3588852>
- Basten, B., Hills, M., Klint, P., Landman, D., Shahi, A., Steindorfer, M., & Vinju, J. (2015, March). M3: A general model for code analytics in rascal. *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. <https://doi.org/10.1109/swan.2015.7070485>
- Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023, March). Programming is hard - or at least it used to be. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education v. 1*. <https://doi.org/10.1145/3545945.3569759>
- Biggs, J., & Collis, K. (1982). *Evaluating the quality of learning: The SOLO taxonomy*. Academic Press, Inc.
- Biggs, J., & Tang, C. (2011). *Teaching for quality learning at university* (4th ed.). McGraw-Hill/Society for Research into Higher Education/Open University Press.
- Bloxham, S., den-Outer, B., Hudson, J., & Price, M. (2015). Let's stop the pretence of consistent marking: Exploring the multiple limitations of assessment criteria. *Assessment & Evaluation in Higher Education*, 41(3), 466–481. <https://doi.org/10.1080/02602938.2015.1024607>
- Blumenfeld, P. C., Soloway, E., Marx, R. W., Krajcik, J. S., Guzdial, M., & Palincsar, A. (1991). Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational Psychologist*, 26(3-4), 369–398. <https://doi.org/10.1080/00461520.1991.9653139>
- Choudhury, R. R., Yin, H., & Fox, A. (2016). Scale-driven automatic hint generation for coding style. In *Intelligent tutoring systems* (pp. 122–132). Springer International Publishing. https://doi.org/10.1007/978-3-319-39583-8_12
- Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7), 638–653. <https://doi.org/10.1109/tse.2002.1019479>
- Fehnker, A., Mader, A., & Rump, A. (2021). Atelier – tutor moderated comments in programming education. In *Technology-enhanced learning for a free, safe,*

- and sustainable world* (pp. 379–383). Springer International Publishing. https://doi.org/10.1007/978-3-030-86436-1_39
- Fournier-Viger, P., Nkambou, R., & Nguifo, E. M. (2010). Building intelligent tutoring systems for ill-defined domains. In *Studies in computational intelligence* (pp. 81–101). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14363-2_5
- Gallagher, S. E., & Savage, T. (2020). Challenge-based learning in higher education: An exploratory literature review. *Teaching in Higher Education*, 1–23. <https://doi.org/10.1080/13562517.2020.1863354>
- Glassman, E. L., Scott, J., Singh, R., Guo, P. J., & Miller, R. C. (2015). OverCode. *ACM Transactions on Computer-Human Interaction*, 22(2), 1–35. <https://doi.org/10.1145/2699751>
- Gross, S., Zhu, X., Hammer, B., & Pinkwart, N. (2012). Cluster based feedback provision strategies in intelligent tutoring systems. In *Intelligent tutoring systems* (pp. 699–700). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-30950-2_127
- Heckel, R. (2006). Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1), 187–198. <https://doi.org/10.1016/j.entcs.2005.12.018>
- Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*. <https://doi.org/10.1145/1930464.1930480>
- Ingalalli, V., Ienco, D., & Poncelet, P. (2016). SuMGra: Querying multigraphs via efficient indexing. In *Lecture notes in computer science* (pp. 387–401). Springer International Publishing. https://doi.org/10.1007/978-3-319-44403-1_24
- Jonsson, A., & Svingby, G. (2007). The use of scoring rubrics: Reliability, validity and educational consequences. *Educational Research Review*, 2(2), 130–144. <https://doi.org/10.1016/j.edurev.2007.05.002>
- Keuning, H., Jeuring, J., & Heeren, B. (2019). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19(1), 1–43. <https://doi.org/10.1145/3231711>
- Klint, P., Storm, T. van der, & Vinju, J. (2011). EASY meta-programming with rascal. In *Lecture notes in computer science* (pp. 222–289). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-18023-1_6
- Krajcik, J. S., & Shin, N. (2014). Project-based learning. In R. K. Sawyer (Ed.), *The cambridge handbook of the learning sciences* (pp. 275–297). Cambridge University Press. <https://doi.org/10.1017/cbo9781139519526.018>
- Krathwohl, D. R. (2002). A revision of bloom’s taxonomy: An overview. *Theory Into Practice*, 41(4), 212–218. https://doi.org/10.1207/s15430421tip4104_2
- Lazar, T., Možina, M., & Bratko, I. (2017). Automatic extraction of AST patterns for debugging student programs. In *Lecture notes in computer science* (pp. 162–174). Springer International Publishing. https://doi.org/10.1007/978-3-319-61425-0_14
- Le, N.-T., Loll, F., & Pinkwart, N. (2013). Operationalizing the continuum between well-defined and ill-defined problems for educational technology. *IEEE Transactions on Learning Technologies*, 6(3), 258–270. <https://doi.org/10.1109/tlt.2013.16>
- Le, N.-T., & Pinkwart, N. (2014, January). Towards a classification for program-

- ming exercises. *Proceedings of the 2nd Workshop on AI-Supported Education for Computer Science*. <https://publications.informatik.hu-berlin.de/archive/cses/publications/Towards-a-Classification-for-Programming-Exercises.pdf>
- Learning tools interoperability (LTI) assignment and grade services specification*. (2019). [Standard]. IMS Global Learning Consortium, Inc. <https://www.imsglobal.org/spec/lti-ags/v2p0/>
- Lönngrén, J. (2017). *Wicked problems in engineering education: Preparing future engineers to work for sustainability* [PhD thesis, Chalmers University of Technology]. <https://publications.lib.chalmers.se/records/fulltext/250857/250857.pdf>
- Mader, A., Fehnker, A., & Dertien, E. (2020). Tinkering in informatics as teaching method. *Proceedings of the 12th International Conference on Computer Supported Education*. <https://doi.org/10.5220/0009467304500457>
- McBroom, J., Koprinska, I., & Yacef, K. (2022). A survey of automated programming hint generation: The HINTS framework. *ACM Computing Surveys*, 54(8), 1–27. <https://doi.org/10.1145/3469885>
- Mens, K., Nijssen, S., & Pham, H.-S. (2021, August). The good, the bad, and the ugly: Mining for patterns in student source code. *Proceedings of the 3rd International Workshop on Education Through Advanced Software Engineering and Artificial Intelligence*. <https://doi.org/10.1145/3472673.3473958>
- Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2023a). *Automated grading and feedback tools for programming education: A systematic review*. arXiv. <https://doi.org/10.48550/ARXIV.2306.11722>
- Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2023b, June). Machine learning-based automated grading and feedback tools for programming: A meta-analysis. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education v. 1*. <https://doi.org/10.1145/3587102.3588822>
- Mitrovic, A. (2011). Fifteen years of constraint-based tutors: What we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, 22(1-2), 39–72. <https://doi.org/10.1007/s11257-011-9105-9>
- Moghadam, J. B., Choudhury, R. R., Yin, H., & Fox, A. (2015, March). Auto-Style. *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*. <https://doi.org/10.1145/2724660.2728672>
- Možina, M., & Lazar, T. (2018). Syntax-based analysis of programming concepts in python. In *Lecture notes in computer science* (pp. 236–240). Springer International Publishing. https://doi.org/10.1007/978-3-319-93846-2_43
- Možina, M., Lazar, T., & Bratko, I. (2018). Identifying typical approaches and errors in prolog programming with argument-based machine learning. *Expert Systems with Applications*, 112, 110–124. <https://doi.org/10.1016/j.eswa.2018.06.029>
- Nguyen, A., Piech, C., Huang, J., & Guibas, L. (2014). Codewebs: Scalable homework search for massive open online programming courses. *Proceedings of the 23rd International Conference on World Wide Web - WWW '14*. <https://doi.org/10.1145/2566486.2568023>
- Nye, B. D., Boyce, M. W., & Sottolare, R. A. (2016). Defining the ill-defined: From abstract principles to applied pedagogy. In R. A. Sottolare, A. C. Graesser, X. Hu, A. Olney, B. Nye, & A. M. Sinatra (Eds.), *Design recommendations for intelligent tutoring systems: Volume 4 - domain modeling*





- (pp. 19–37). Army Research Laboratory.
- Paiva, J. C., Leal, J. P., & Figueira, Á. (2022). Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education*. <https://doi.org/10.1145/3513140>
- Patidar, A., & Suman, U. (2015). A survey on software architecture evaluation methods. *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 967–972. <https://ieeexplore.ieee.org/document/7100391>
- Rump, A., Fehnker, A., & Mader, A. (2021). Automated assessment of learning objectives in programming assignments. In C. Cristea Alexandra I. and Troussas (Ed.), *Intelligent tutoring systems* (pp. 299–309). Springer International Publishing. https://doi.org/10.1007/978-3-030-80421-3_33
- Rump, A., & Zaytsev, V. (2022, October). A refined model of ill-definedness in project-based learning. *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. <https://doi.org/10.1145/3550356.3556505>
- Simon, H. A. (1978). Information-processing theory of human problem solving. In W. Estes (Ed.), *Handbook of learning and cognitive processes (volume 5)*. Psychology Press. <https://doi.org/10.4324/9781315770314-14>
- Suraweera, P., Mitrovic, A., & Martin, B. (2005). A knowledge acquisition system for constraint-based intelligent tutoring systems. In C.-K. Looi, G. I. McCalla, B. Bredeweg, & J. Breuker (Eds.), *Artificial intelligence in education - supporting learning through intelligent and socially informed technology, proceedings of the 12th international conference on artificial intelligence in education, AIED 2005, july 18-22, 2005, amsterdam, the netherlands* (Vol. 125, pp. 638–645). IOS Press.
- Xu, S., & Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4), 360–384. <https://doi.org/10.1109/tse.2003.1191799>
- Yan, X., & Han, J. (2002). gSpan: Graph-based substructure pattern mining. *2002 IEEE International Conference on Data Mining, 2002. Proceedings*. <https://doi.org/10.1109/icdm.2002.1184038>
- Zhang, A. G., Chen, Y., & Oney, S. (2023, April). VizProg: Identifying misunderstandings by visualizing students' coding progress. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. <https://doi.org/10.1145/3544548.3581516>

Appendices

Appendix A

Criteria

This appendix contains a flat list of criteria for the Algorithms in Creative Technology (AiC) and Software Systems (SS) courses. For each criterion we also list the source of that criterion (project description or rubric, which section or category) and the result of trying to configure that criterion in our prototype. These results are indicated with four icons, followed by a short explanation and the pattern if the criterion is configurable in the tool. The icons have the following meaning:

-  This criterion is implemented
-  This criterion could not be implemented with the TypeGraph, but is likely doable with a PDG
-  This criterion could not be implemented with the TypeGraph, and it remains to be seen if it is possible with another program representation
-  This criterion is not assessable from the code, using static analysis, within this framework or is hard for other reasons

A.1 Algorithms in Creative Technology

1. Combines at least 3 from the 4 topics in one application.

Source: Description/Scope

-  Not in this framework, combines results from multiple criteria

2. Includes meaningful randomness (normal distribution and Perlin noise).

Source: Description/Scope

-  Detect usage of randomness

```
var randomMethods = [  
  "java+method:///processing/core/PApplet/random(  
    float,float)",  
  "java+method:///processing/core/PApplet/random(  
    float)",
```

```

"java+method:///processing/core/PApplet/
  randomGaussian()",
"java+method:///processing/core/PApplet/noise(float
)",
"java+method:///processing/core/PApplet/noise(float
, float)",
"java+method:///processing/core/PApplet/noise(float
, float, float)"
]

var criterion = {
  "criterion": "Includes meaningful randomness (
    normal distribution and Perlin noise).",
  "patterns": randomMethods.map(randomMethod => ({
    "verdict": "positive",
    "pattern": [
      [ "method", { "annotation": { "scheme": "java+
        method" } }, "method" ],
      [ "method", "invokes", "randomMethod" ],
      [ "randomMethod", { "annotation": { "location":
        randomMethod } }, "randomMethod" ]
    ]
  )))
}

```

✗ Determine if its meaningful

- Includes use of forces, with difficulty at least equivalent to catapult, billiard or mass-spring-damper systems.

Source: Description/Scope

~ Not with the TypeGraph, but a PDG could be used to match calculations with force vectors

- Includes particles.

Source: Description/Scope

✓ Particle systems have a rather well-defined class structure

```

{
  "criterion": "Includes particles.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "Particle", { "annotation": { "scheme": "
          java+class" } }, "Particle" ],
        [ "Particle", "contains", "lifespan" ],
        [ "lifespan", { "annotation": { "scheme": "
          java+field" } }, "lifespan" ],
        [ "Particle", "contains", "update" ],

```

```

    [ "update", { "annotation": { "scheme": "java
      +method" } }, "update" ],
    [ "update", "accessesField", "lifespan" ],
    [ "Particle", "contains", "isDead" ],
    [ "isDead", { "annotation": { "scheme": "java
      +method" } }, "isDead" ],
    [ "isDead", "accessesField", "lifespan" ],
    [ "isDead", "dependsOn", "boolean" ],
    [ "boolean", { "annotation": { "location": "
      java+primitiveType:///boolean" } }, "
      boolean" ],
    [ "ParticleSystem", { "annotation": { "scheme
      ": "java+class" } }, "ParticleSystem" ],
    [ "ParticleSystem", "contains", "particles"
      ],
    [ "particles", { "annotation": { "scheme": "
      java+field" } }, "particles" ],
    [ "particles", "dependsOn", "Particle" ],
    [ "particles", "dependsOn", "ArrayList" ],
    [ "ArrayList", { "annotation": { "location": "
      java+class:///java/util/ArrayList" } },
      "ArrayList" ],
    [ "ParticleSystem", "contains", "run" ],
    [ "run", "accessesField", "particles" ],
    [ "run", "invokes", "isDead" ]
  ],
  "children": [
    {
      "verdict": "neutral",
      "pattern": [
        [ "isDead", "contains", "someParam" ],
        [ "someParam", { "annotation": { "scheme
          ": "java+parameter" } }, "someParam"
        ]
      ]
    }
  ]
}

```

5. Includes flocking.

Source: Description/Scope

✓ Similar to a particle system, flocking uses a common class structure

```

{
  "criterion": "Includes flocking.",
  "patterns": [
    {

```

```

    "verdict": "positive",
    "pattern": [
      [ "Flock", { "annotation": { "scheme": "java+
        class" } }, "Flock" ],
      [ "Flock", "dependsOn", "Boid" ],
      [ "Flock", "contains", "run" ],
      [ "run", { "annotation": { "scheme": "java+
        method" } }, "run" ],
      [ "run", "invokes", "flockingMethod" ],
      [ "Boid", { "annotation": { "scheme": "java+
        class" } }, "Boid" ],
      [ "Boid", "contains", "flockingMethod" ],
      [ "flockingMethod", { "annotation": { "scheme
        ": "java+method" } }, "flockingMethod" ],
      [ "flockingMethod", "contains", "boidsParam"
        ],
      [ "boidsParam", { "annotation": { "scheme": "
        java+parameter" } }, "boidsParam" ],
      [ "boidsParam", "dependsOn", "ArrayList" ],
      [ "ArrayList", { "annotation": { "location":
        "java+class:///java/util/ArrayList" } },
        "ArrayList" ],
      [ "boidsParam", "dependsOn", "Boid" ]
    ]
  }
]
}

```

- At least two (nontrivial) classes have to be written by yourself.

Source: Description/Understanding

✘ Not with the TypeGraph, but on more detailed graphs, patterns of a few common sources could go a long way

- Flocking and particles should not both be from Shiffman (or other sources on the web).

Source: Description/Understanding

✘ Similar to 6

- A major part of significant complexity has to be written by yourself.

Source: Description/Understanding

✘ Similar to 6

- Quote the sources that you use.

Source: Description/Understanding

✘ Comments are not in the TypeGraph, but detecting links or mentions of common sources should be doable

10. How much interaction is there with the user.

Source: Description/Complexity

✔ See 17 for patterns that can be used with this metric

✘ Metrics are not implemented in the prototype

11. How much interaction is between the classes. Does an event in one class have an effect on objects of other class?

Source: Description/Complexity

✔ Finds method calls between classes, but ignoring method calls from the main tab

```
{
  "criterion": "How much interaction is between the
    classes.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "class1", { "annotation": { "scheme": "java
          +class" } }, "class1" ],
        [ "class2", { "annotation": { "scheme": "java
          +class" } }, "class2" ],
        [ "method2", { "annotation": { "scheme": "
          java+method" } }, "method2" ],
        [ "class1", "invokes", "method2" ]
      ],
      "children": [
        {
          "verdict": "neutral",
          "pattern": [
            [ "PApplet", { "annotation": { "location
              ": "java+class:///processing/core/
              PApplet" } }, "PApplet" ],
            [ "class1", "extends", "PApplet" ]
          ]
        }
      ]
    }
  ]
}
```

✘ Metrics are not implemented in the prototype

12. How many classes are there?

Source: Description/Complexity

✔ Counting classes is simple

```
{
```



```

"critierion": "How many classes are there?",
"patterns": [
  {
    "verdict": "positive",
    "pattern": [
      [ "class", { "annotation": { "scheme": "java+
class" } }, "class" ]
    ]
  }
]
}

```

✗ Metrics are not implemented in the prototype

13. Keep the main draw function as small as possible.

Source: Description/Style

~ Not with the TypeGraph, but might be doable with a PDG

14. Do not use unnecessary global variables, especially not for value passing.

Source: Description/Style + Rubric/Structure

✓ Find global variables which are accessed from other classes. Constants are fine.

```

{
  "critierion": "Do not use unnecessary global
variables, especially not for value passing.",
  "patterns": [
    {
      "verdict": "negative",
      "pattern": [
        [ "PApplet", { "annotation": { "location": "
java+class:///processing/core/PApplet" }
}, "PApplet" ],
        [ "mainTab", { "annotation": { "scheme": "
java+class" } }, "mainTab" ],
        [ "mainTab", "extends", "PApplet" ],
        [ "globalVar", { "annotation": { "scheme": "
java+field" } }, "globalVar" ],
        [ "mainTab", "contains", "globalVar" ],
        [ "otherClass", { "annotation": { "scheme": "
java+class" } }, "otherClass" ],
        [ "otherClass", "accessesField", "globalVar"
]
      ],
      "children": [
        {
          "verdict": "neutral",
          "pattern": [

```

```

        [ "globalVar", { "annotation": { "
            modifier": "final" } }, "globalVar" ]
    ]
}
]
}
]
}

```

~ Detecting value passing requires detecting modifications to global variables from classes, which is not in the TypeGraph, but might be doable with a PDG

15. Have constants for your program as global variables (e.g. number of players, etc.), not hidden in classes.

Source: Description/Style + Rubric/Structure

~ Not with the TypeGraph, because we can't detect constants (see previous criterion also), but could be doable with a PDG

16. Do not hard code parameters that may change during the program development. Also do not hardcode graphical elements that might change.

Source: Description/Style + Rubric/Structure

~ Not with the TypeGraph, because it doesn't include literals, but could be done with a graph that does include those

17. Do not hide user interaction in classes. Deal with user interaction in the following way: ...

Source: Description/Style + Rubric/Structure

✓ Find the usage of event handling methods as positive examples and the usage of Processing's event variables outside those methods as negative examples

```

var criterion = {
  "criterion": "Do not hide user interaction in
    classes.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "PApplet", { "annotation": { "location": "
            java+class:///processing/core/PApplet" }
          }, "PApplet" ],
        [ "mainTab", { "annotation": { "scheme": "
            java+class" } }, "mainTab" ],
        [ "mainTab", "extends", "PApplet" ],
        [ "mainTab", "contains", "method" ],
        [ "method", { "annotation": { "scheme": "java
            +method" } }, "method" ],
        [ "method", "overrides", "PApplet.method" ]
      ]
    }
  ]
}

```

```

],
"children": [
  {
    "verdict": "neutral",
    "pattern": [
      [ "PApplet.method", { "annotation": { "
        location": "java+method:///processing
        /core/PApplet/draw()" } }, "PApplet.
        method" ]
    ]
  },
  {
    "verdict": "neutral",
    "pattern": [
      [ "PApplet.method", { "annotation": { "
        location": "java+method:///processing
        /core/PApplet/setup()" } }, "PApplet.
        method" ]
    ]
  },
  {
    "verdict": "neutral",
    "pattern": [
      [ "PApplet.method", { "annotation": { "
        location": "java+method:///processing
        /core/PApplet/settings()" } }, "
        PApplet.method" ]
    ]
  }
],
},
{
  "verdict": "neutral",
  "pattern": [
    [ "method", "accessesField", "field" ],
    [ "method", { "annotation": { "scheme": "java
      +method" } }, "method" ]
  ],
  "children":
    [ "key", "keyCode", "keyPressed", "
      mouseButton", "mousePressed", "mouseX", "
      mouseY", "pmouseX", "pmouseY" ].map(field
      => (
        {
          "verdict": "negative",
          "pattern": [
            [ "field", { "annotation": { "location
              ": `java+field:///processing/core/
              PApplet/${field}` } }, "field" ]
          ],
        }
      )
    ),

```

```

"children":
  [ "keyPressed()", "keyReleased()", "
    keyPressed()", "mouseClicked()", "
    mouseDragged()", "mouseMoved()", "
    mousePressed()", "mouseReleased()",
    "mouseWheel()" ].map(method => (
  {
    "verdict": "neutral",
    "pattern": [
      [ "PApplet", { "annotation": { "
        location": "java+class:///
        processing/core/PApplet" } },
        "PApplet" ],
      [ "mainTab", { "annotation": { "
        scheme": "java+class" } }, "
        mainTab" ],
      [ "mainTab", "extends", "PApplet"
        ],
      [ "mainTab", "contains", "method"
        ],
      [ "method", { "annotation": { "
        scheme": "java+method" } }, "
        method" ],
      [ "method", "overrides", "PApplet
        .method" ],
      [ "PApplet.method", { "annotation
        ": { "location": `java+method
        ::///processing/core/PApplet/$
        {method}` } }, "PApplet.
        method" ]
    ]
  }
  ))
}
))
}
]
}

```

18. Have things that belong together logically, in one class.

Source: Description/Style + Rubric/Structure

✘ Not with the TypeGraph, depends on content

19. When classes or methods become too large, split them in a logical way.

Source: Description/Style + Rubric/Structure

✘ Not with the TypeGraph, but with a different graph large methods and classes could be flagged

20. Use arrays (and loops) when you have similar values.

Source: Description/Style + Rubric/Structure

~ Not with the TypeGraph, but could be doable with a PDG

21. Use functions when you have similar code.

Source: Description/Style + Rubric/Structure

✓ Positive examples: functions that are reused in multiple places

```
{
  "criterion": "Use functions when you have similar
    code.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "reusedMethod", { "annotation": { "scheme":
          "java+method" } }, "reusedMethod" ],
        [ "caller1", "invokes", "reusedMethod" ],
        [ "caller1", { "annotation": { "scheme": "
          java+method" } }, "caller1" ],
        [ "caller2", "invokes", "reusedMethod" ],
        [ "caller2", { "annotation": { "scheme": "
          java+method" } }, "caller2" ]
      ]
    }
  ]
}
```

✗ Finding duplicate code is not supported in the framework

22. Do not have unused variables.

Source: Description/Style + Rubric/Style

~ Not with the TypeGraph (it includes variables, but not variable-usage), but could be done with PDG

23. Do not have unused code.

Source: Description/Style + Rubric/Style

✓ Find unused methods

```
{
  "criterion": "Do not have unused code",
  "patterns": [
    {
      "verdict": "negative",
      "pattern": [
        [ "method", { "annotation": { "scheme": "java
          +method" } }, "method" ]
      ],
      "children": [
        {
```

```

        "verdict": "neutral",
        "pattern": [
          [ "other", "invokes", "method" ]
        ]
      },
      {
        "verdict": "neutral",
        "pattern": [
          [ "method", "overrides", "other" ]
        ]
      },
      {
        "verdict": "neutral",
        "pattern": [
          [
            "PApplet.main",
            { "annotation": { "location": "java+
              method:///processing/core/PApplet/
                main(java.lang.String%5B%5D)" } },
            "PApplet.main"
          ],
          [ "method", "invokes", "PApplet.main" ]
        ]
      }
    ]
  }
]
}

```

~ Unreachable code (due to false if-statements or loops) is not detectable in the TypeGraph, but could be done with a PDG

24. Have comments and a header.

Source: Description/Style + Rubric/Documentation

✗ Comments are not in the TypeGraph, but could be done with a different representation

25. Functional correctness.

Source: Rubric

✗ A free criterion, not automatically assessable in general. It might be possible to detect some common mistakes to possibly catch some cases, but not with the TypeGraph

26. Efficiency

Source: Rubric

✗ Similar to 25, too broad for general assessment, but there might be some common mistakes that could be detected

27. Attention for details.

Source: Rubric/Appearance

✗ Too broad, and probably not assessable in code

A.2 Software Systems

1. A standard game can be played on both client and server in conjunction with the reference server and client, respectively.

Source: Description/Functional Requirements

✗ Not with static code analysis

2. The client can play as a human player, controlled by the user.

Source: Description/Functional Requirements

✗ Not with the TypeGraph, maybe with static code analysis

3. The client can play as a computer player, controlled by AI.

Source: Description/Functional Requirements + Rubric/Functionality/Client

✗ Not with the TypeGraph, maybe with static code analysis

4. When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.

Source: Description/Functional Requirements + Rubric/Functionality/Server

~ Not with the TypeGraph, but should be doable with PDG: the method to start listening on a port is known, so we could check if the provided parameter is a variable and not a literal or constant.

5. When the client is started, it should ask the user for the IP-address and port number of the server to connect to.

Source: Description/Functional Requirements + Rubric/Functionality/Client

~ Not with the TypeGraph, but should be doable with PDG: two variables are assigned through readline methods, which are passed to the connection method.

6. When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.

Source: Description/Functional Requirements + Rubric/Functionality/Client

✗ Not with the TypeGraph, maybe with static code analysis

7. The client can play a full game automatically as the AI without intervention by the user.

Source: Description/Functional Requirements + Rubric/Functionality/Client

✘ Not with the TypeGraph, maybe with static code analysis

8. The AI difficulty can be adjusted by the user via the TUI.

Source: Description/Functional Requirements

✘ Not with the TypeGraph, maybe with static code analysis

9. All communication outside of playing a game, such as handshakes and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively.

Source: Description/Functional Requirements

✘ Not with static code analysis

10. Whenever a client loses connection to a server, the client should gracefully terminate.

Source: Description/Functional Requirements + Rubric/Functionality/Networking + Rubric/Functionality/Client

~ Not with the TypeGraph, but could be doable with a PDG (to find the place where detection loss is connected, the appropriate exception is handled)

11. Whenever a client disconnects during a game, the server should inform the other clients and end the game, allowing the other player to start a new game.

Source: Description/Functional Requirements + Rubric/Functionality/Networking + Rubric/Functionality/Server

~ Extension of 10

12. Possible extension: allow players to send short pieces of text to a global chat or to individual players.

Source: Description/Extensions

✘ Not with the TypeGraph, maybe with static code analysis

13. Chatbox extension adheres to the protocol.

Source: Description/Extensions

✘ Not with the TypeGraph, maybe with static code analysis

14. Possible extension: maintain a player ranking.

Source: Description/Extensions

✘ Not with the TypeGraph, maybe with static code analysis

15. The player ranking works as designed, the chosen statistic is implemented correctly.

Source: Description/Extensions

- ✗ Not with the TypeGraph, maybe with static code analysis
- 16. Possible extension: authenticate a player using public-/private-key algorithms.
 - Source: Description/Extensions
 - ✗ Not with the TypeGraph, maybe with static code analysis
- 17. Authentication extension adheres to the protocol.
 - Source: Description/Extensions
 - ✗ Not with the TypeGraph, maybe with static code analysis
- 18. Possible extension: encrypt communication between client and server.
 - Source: Description/Extensions
 - ✗ Not with the TypeGraph, maybe with static code analysis
- 19. Encryption extension adheres to key negotiation protocol.
 - Source: Description/Extensions
 - ✗ Not with the TypeGraph, maybe with static code analysis
- 20. All game rules are implemented with reasonable quality.
 - Source: Description/Midway submission
 - ✗ Not with the TypeGraph, maybe with static code analysis
- 21. All public classes and methods of the game logic have Javadoc with reasonable quality.
 - Source: Description/Midway submission
 - ✗ Not with the TypeGraph (Javadoc is not present in the TypeGraph), but the presence could be detected with a different code representation. Judging “reasonable quality” will be more difficult.
- 22. Complex methods of the game logic have comments with reasonable quality.
 - Source: Description/Midway submission
 - ✗ Comments are not in the TypeGraph
- 23. There are unit tests for the game logic, including at least testing whether a move is performed correctly, testing a gameover condition, and testing a random play of a full game from start to finish including checking the gameover condition, with reasonable quality.
 - Source: Description/Midway submission
 - ✗ Hard to detect, because we need to identify what classes handle the game logic (and the separation is usually not that clear)
- 24. There is an initial version of the report including a section on the overall testing strategy with incorporation of coverage metrics of the game logic and an explanation of the test plan for the game logic, with reasonable quality.

- Source: Description/Midway submission
- ✗ Not in the code
25. The game rules are implemented correctly.
- Source: Rubric/Functionality/Game logic
- ✗ Not with the TypeGraph, maybe with static code analysis
26. Many games can be played independently.
- Source: Rubric/Functionality/Game logic
- ✗ Not with the TypeGraph, maybe with static code analysis
27. Game logic methods do not crash with unchecked exceptions from incorrect usage by other classes (invalid moves, negative indices, etc.)
- Source: Rubric/Functionality/Game logic
- ✗ Not with static code analysis (at least not with patterns, gets close to correctness proofs)
28. Game logic is thread-safe (no race conditions even when used from multiple threads)
- Source: Rubric/Functionality/Game logic
- ✗ Not with the TypeGraph, maybe with static code analysis
29. Networking is implemented correctly, according to the protocol.
- Source: Rubric/Functionality/Networking
- ✗ Not with the TypeGraph, maybe with static code analysis
30. Incorrect usage by client or server code does not result in protocol violations.
- Source: Rubric/Functionality/Networking
- ✗ Not with the TypeGraph, maybe with static code analysis
31. A malicious attacker cannot make the software crash by sending bad messages.
- Source: Rubric/Functionality/Networking
- ✗ Not with static code analysis (at least not with patterns, gets close to correctness proofs)
32. A PING message always results in an immediate PONG reply, even if the client or server code is busy with earlier messages.
- Source: Rubric/Functionality/Networking
- ✗ Not with the TypeGraph, maybe with static code analysis
33. The networking code is thread-safe.
- Source: Rubric/Functionality/Networking
- ✗ Not with the TypeGraph, maybe with static code analysis

34. The client is user friendly.
Source: Rubric/Functionality/Client
✗ Not with static code analysis
35. The UI does not crash on malformed input.
Source: Rubric/Functionality/Client
✗ Not with static code analysis (at least not with patterns, gets close to correctness proofs)
36. After playing a game, it is possible to play another game without reconnecting.
Source: Rubric/Functionality/Client
✗ Not with the TypeGraph, maybe with static code analysis
37. The user can always request a list of allowed commands and the current status.
Source: Rubric/Functionality/Client
✗ Not with the TypeGraph, maybe with static code analysis
38. The client can run in automatic mode, queueing for games and playing as the AI at a chosen difficulty level until asked to stop.
Source: Rubric/Functionality/Client
✗ Not with the TypeGraph, maybe with static code analysis
39. Multiple games can be played simultaneously.
Source: Rubric/Functionality/Server
✗ Not with the TypeGraph, maybe with static code analysis
40. A client can play multiple games after each other.
Source: Rubric/Functionality/Server
✗ Not with the TypeGraph, maybe with static code analysis
41. The server is clearly thread-safe and games progress independent of each other.
Source: Rubric/Functionality/Server
✗ Not with the TypeGraph, maybe with static code analysis
42. The server maintains a list of online users and active games and these are thread-safe.
Source: Rubric/Functionality/Server
✗ Not with the TypeGraph, maybe with static code analysis
43. When players disconnect and games end, references to related objects are correctly removed.
Source: Rubric/Functionality/Server

- ~ Not with TypeGraph, but might be doable with a PDG
44. There are no compilation errors or failed tests.
Source: Rubric/Software/Packaging
✗ Not in this framework (requires compiling and running tests)
 45. The project builds and can be tested without errors.
Source: Rubric/Software/Packaging
✗ Not in this framework (requires compiling and running tests)
 46. Any required libraries other than junit5 are included.
Source: Rubric/Software/Packaging
✗ Not with the TypeGraph, maybe with static code analysis
 47. The README file is present and contains specific instructions on building, testing and running the software.
Source: Rubric/Software/Packaging
✗ Not with static code analysis
 48. The Javadoc is exported as HTML pages in a directory structure separate from the sources.
Source: Rubric/Software/Packaging
✗ Not in this framework (requires more than just the code)
 49. No Checkstyle violations.
Source: Rubric/Software/Clean code
✗ Not in this framework (requires running Checkstyle)
 50. Constants are used where appropriate.
Source: Rubric/Software/Clean code
✓ We can find constants

```
{
  "criterion": "Constants are used where appropriate
  .",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "field", { "annotation": { "scheme": "java+
          field" } }, "field" ],
        [ "field", { "annotation": { "modifier": "
          final" } }, "field" ]
      ]
    }
  ]
}
```


59. The server is tested with unit tests that check whether the protocol is handled correctly.

Source: Rubric/Software/Test

✔ Simplified: the server is tested with unit tests

```
var criterion = {
  "criterion": "The server is tested with unit tests
    that check whether the protocol is handled
    correctly.",
  "patterns": [ "java+class:///Test", "java+interface
    ::///org/junit/jupiter/api/Test" ].map(test => (
    {
      "verdict": "positive",
      "pattern": [
        [ "server", { "annotation": { "scheme": "java
          +class" } }, "server" ],
        [ "ServerSocket", { "annotation": { "location
          ": "java+class:///java/net/ServerSocket"
          } }, "ServerSocket" ],
        [ "server", "dependsOn", "ServerSocket" ],
        [ "serverMethod", { "annotation": { "scheme":
          "java+method" } }, "serverMethod" ],
        [ "server", "contains", "serverMethod" ],
        [ "testMethod", { "annotation": { "scheme": "
          java+method" } }, "testMethod" ],
        [ "testMethod", "dependsOn", "@Test"],
        [ "@Test", { "annotation": { "location": test
          } }, "@Test" ],
        [ "testMethod", "invokes", "serverMethod" ]
      ]
    }
  )
)
```

✘ Protocol handling not with the TypeGraph, maybe with static code analysis

60. The server is tested with an automated test to play several simultaneous games on the server.

Source: Rubric/Software/Test

✘ Not with the TypeGraph, maybe with static code analysis

61. Classes do not access the state of other classes directly.

Source: Rubric/Software/Encapsulation

✔ Check if a class accesses a (non-final) field

```
{
  "criterion": "Classes do not access the state of
    other classes directly.",
```

```

"patterns": [
  {
    "verdict": "negative",
    "pattern": [
      [ "otherClass", { "annotation": { "scheme": "
        java+class" } }, "otherClass" ],
      [ "otherClass", "contains", "otherClassField"
        ],
      [ "otherClassField", { "annotation": { "
        scheme": "java+field" } }, "
        otherClassField" ],
      [ "class", { "annotation": { "scheme": "java+
        class" } }, "class" ],
      [ "class", "accessesField", "otherClassField"
        ]
    ],
    "children": [
      {
        "verdict": "neutral",
        "pattern": [
          [ "otherClassField", { "annotation": { "
            modifier": "final" } }, "
            otherClassField" ]
        ]
      }
    ]
  }
]
}

```

62. All fields except constants are private.

Source: Rubric/Software/Encapsulation

Check for private and final modifiers

```

{
  "criterion": "All fields except constants are
    private.",
  "patterns": [
    {
      "verdict": "negative",
      "pattern": [
        [ "field", { "annotation": { "scheme": "java+
          field" } }, "field" ]
      ],
      "children": [
        {
          "verdict": "positive",
          "pattern": [
            [ "field", { "annotation": { "modifier":

```



```

        "pattern": [
          [ "other", "invokes", "method" ]
        ]
      },
      {
        "verdict": "neutral",
        "pattern": [
          [ "method", { "annotation": { "modifier": "static" } }, "method" ],
          [ "method", "dependsOn", "void" ],
          [ "void", { "annotation": { "location": "java+primitiveType:///void" } }, "void" ],
          [ "method", "contains", "param" ],
          [ "param", { "annotation": { "scheme": "java+parameter" } }, "param" ],
          [ "param", "dependsOn", "stringArray" ],
          [ "stringArray", { "annotation": { "location": "java+array:///java/lang/String%5B%5D" } }, "stringArray" ]
        ]
      },
      {
        "verdict": "neutral",
        "pattern": [
          [ "method", "dependsOn", "@Override" ],
          [ "@Override", { "annotation": { "location": "java+interface:///java/lang/Override" } }, "@Override" ]
        ]
      }
    ].concat(
      [ "java+class:///Test",
        "java+class:///BeforeEach",
        "java+interface:///org/junit/jupiter/api/Test",
        "java+interface:///org/junit/jupiter/api/BeforeEach" ].map(testAttr => (
      {
        "verdict": "neutral",
        "pattern": [
          [ "method", "dependsOn", "@Test" ],
          [ "@Test", { "annotation": { "location": testAttr } }, "@Test" ]
        ]
      }
    )))
  }
}

```

64. Other classes cannot directly manipulate fields managed by an object via getters/setters.

Source: Rubric/Software/Encapsulation

~ Not with the TypeGraph. Detecting getters and setters would be more appropriate with a PDG, which can really check if a method does nothing more than returning a field or assigning to a field.

65. There is a clear separation of concerns between classes.

Source: Rubric/Software/Structure

✗ Hard to identify the concerns that are addressed in classes automatically

66. There is more than one component (package).

Source: Rubric/Software/Structure

✓ We only mark the negatives here, if there is only one package. Marking two packages as positive, would result in a spamming of results for every pair of packages in a project.

```
{
  "criterion": "There is more than one component (
    package).",
  "patterns": [
    {
      "verdict": "negative",
      "pattern": [
        [ "package", { "annotation": { "scheme": "
          java+package" } }, "package" ]
      ],
      "children": [
        {
          "verdict": "neutral",
          "pattern": [
            [ "otherPackage", { "annotation": { "
              scheme": "java+package" } }, "
              otherPackage" ]
          ]
        }
      ]
    }
  ]
}
```

67. There are clearly defined components, reflected in the package structure.

Source: Rubric/Software/Structure

✗ This may be doable through connectivity within packages, but that is not possible in this framework

68. Clear and logical dependencies between classes and between components.

Source: Rubric/Software/Structure

✗ Too subjective (though is specified more in the next few criteria)

69. Interfaces are applied where appropriate to ensure low coupling between components.

Source: Rubric/Software/Structure

✓ Translated to: if a class implements an interface, then there should be no fields or parameters that depend directly on that class

```
{
  "criterion": "Interfaces are applied where
    appropriate to ensure low coupling between
    components.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "interface", { "annotation": { "scheme": "
          java+interface" } }, "interface" ],
        [ "class", { "annotation": { "scheme": "java+
          class" } }, "class" ],
        [ "class", "implements", "interface" ]
      ],
      "children": [
        {
          "verdict": "negative",
          "pattern": [
            [ "field", { "annotation": { "scheme": "
              java+field" } }, "field" ],
            [ "field", "dependsOn", "class" ]
          ]
        },
        {
          "verdict": "negative",
          "pattern": [
            [ "param", { "annotation": { "scheme": "
              java+parameter" } }, "param" ],
            [ "param", "dependsOn", "class" ]
          ]
        }
      ]
    }
  ]
}
```

✗ Detecting class usage across package boundaries results in many false negatives, because packages do not always fully reflect components (see 67)

70. The product is easily extendable by hiding implementations behind interfaces at appropriate abstraction levels, such that new implementations can easily be added without requiring extensive changes in the rest of the code.
- Source: Rubric/Software/Structure
- ✔ ✗ Similar to 69
71. In particular, the network protocol can be replaced without affecting the rest of the code.
- Source: Rubric/Software/Structure
- ✗ Hard to detect, because it requires understanding of components and which component handles the network
72. The MVC pattern is applied such that the UI code is separated from the rest of the code.
- Source: Rubric/Software/Design Patterns
- ✗ Hard to detect, because the implementations rarely provide perfect separation. It is hard to detect which class is a view class, which is a controller etc.
73. A Listener pattern is used appropriately.
- Source: Rubric/Software/Design Patterns
- ? Unclear what does and does not qualify as a Listener in this project
74. The UI is separated through interfaces such that other components do not know the concrete UI classes.
- Source: Rubric/Software/Design Patterns
- ✗ Hard to detect, because we need to identify which are the UI components (and the separation is usually not that clear)
75. The game logic is not aware of any concrete classes of the rest of the program.
- Source: Rubric/Software/Design Patterns
- ✗ Hard to detect, because we need to identify the game logic components
76. The UI is separated such that most components are not even aware of the existence of a UI.
- Source: Rubric/Software/Design Patterns
- ✗ Hard to detect, because it depends on identifying components
77. A creational pattern such as the Factory pattern or dependency injection is used appropriately.
- Source: Rubric/Software/Design Patterns
- ~ [Factory] Not with the TypeGraph, because it does not explicitly include the return type of methods. If it did, this might be a reasonable pattern:

```
[
  [ "factory", { "annotation": { "scheme": "java+
    class" } }, "factory" ],
  [ "factoryMethod", { "annotation": { "scheme": "
    java+method" } }, "factoryMethod" ],
  [ "factoryMethod", { "annotation": { "modifier": "
    public" } }, "factoryMethod" ],
  [ "factory", "contains", "factoryMethod" ],
  [ "product", { "annotation": { "scheme": "java+
    class" } }, "product" ],
  [ "productConstructor", { "annotation": { "scheme":
    "java+constructor" } }, "productConstructor"
  ],
  [ "product", "contains", "productConstructor" ],
  [ "factoryMethod", "returns", "product" ],
  [ "factoryMethod", "invokes", "productConstructor"
  ]
]
```

Still, a PDG would be better suited to check that the returned value is actually the constructed value.

~ [Dependency Injection] Doable with the TypeGraph with reasonable results, but with a PDG you could make sure the parameter is actually assigned to that field with the same type (also takes out the duplicates when multiple instances of the same type are injected)

```
{
  "criterion": "A creational pattern such as the
    Factory pattern or dependency injection is used
    appropriately.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "injectee", { "annotation": { "scheme": "
          java+class" } }, "injectee" ],
        [ "injecteeConstructor", { "annotation": { "
          scheme": "java+constructor" } }, "
          injecteeConstructor" ],
        [ "injectee", "contains", "
          injecteeConstructor" ],
        [ "injecteeParam", { "annotation": { "scheme
          ": "java+parameter" } }, "injecteeParam"
        ],
        [ "injecteeConstructor", "contains", "
          injecteeParam" ],
        [ "injecteeField", { "annotation": { "scheme
          ": "java+field" } }, "injecteeField" ],
        [ "injectee", "contains", "injecteeField" ],

```

```

        [ "injecteeParam", "dependsOn", "injectedType
          " ],
        [ "injecteeField", "dependsOn", "injectedType
          " ]
      ]
    }
  ]
}

```

78. Custom exceptions are defined where appropriate and no equivalent pre-defined exceptions exist.

Source: Rubric/Software/Exceptions

✔ Find custom exceptions

```

{
  "criterion": "Custom exceptions are defined where
    appropriate and no equivalent predefined
    exceptions exist.",
  "patterns": [
    {
      "verdict": "positive",
      "pattern": [
        [ "Exception", { "annotation": { "location":
          "java+class:///java/lang/Exception" } },
          "Exception" ],
        [ "customException", { "annotation": { "
          scheme": "java+class" } }, "
          customException" ],
        [ "customException", "extends", "Exception" ]
      ]
    },
    {
      "verdict": "positive",
      "pattern": [
        [ "Exception", { "annotation": { "location":
          "java+class:///java/lang/Exception" } },
          "Exception" ],
        [ "Exception'", "extends", "Exception" ],
        [ "customException", { "annotation": { "
          scheme": "java+class" } }, "
          customException" ],
        [ "customException", "extends", "Exception'"
          ]
      ]
    }
  ]
}

```

✘ Appropriateness of custom exceptions depends too much on under-

standing of the concepts

79. Exceptions are caught and handled appropriately.

Source: Rubric/Software/Exceptions

~ Not with the TypeGraph, but might be doable with a PDG

80. Predefined exceptions are used where appropriate and no predefined exceptions are used where custom exceptions are desired.

Source: Rubric/Software/Exceptions

✗ Similar to 78, exception handling not in the TypeGraph and appropriateness is hard to judge automatically

81. Classes and methods have Javadoc of respectable quality, that is, a concise and complete description for other programmers what a class is for or what a method does, and the possible return values/exceptions.

Source: Rubric/Software/Javadoc

✗ Not with the TypeGraph (Javadoc is not present in the TypeGraph), but the presence could be detected with a different code representation, as well as completeness regarding return values and exceptions. Nevertheless, checking if a description is concise and complete remains hard.

82. Methods of game logic have reasonable nontrivial pre- and postconditions. (sensible ensures/requires statements; full modeling of the game logic is not required)

Source: Rubric/Software/Pre-postconditions

✗ Not in the TypeGraph, requires special JML support (which could be done using a custom graph)

83. Comments are used to aid other programmers in the understanding of the code, and document all non-trivial or implicit implementation details, without adding too much verbosity.

Source: Rubric/Software/Comments

✗ Comments are not in the TypeGraph, but presence could be detected with other graphs. Whether or not they are helpful or too verbose, remains hard to judge.

Appendix B

Analysis results

Tables B.1 and B.2 show the results for the comparison between the assessment results reported from our prototype and the assessment made by a human assessor. *Agree* or *Disagree* means agreement or disagreement between the tool results and the filled in rubric, respectively. *Rubric-not-mentioned* means that a criterion was not mentioned in the filled in rubric, neither could its assessment be derived from the score for that category in the rubric.

In AiC, table B.1, we include the number of occurrences for certain criteria that refer to an amount as to which something happens. We also record *Rubric-positive* or *Rubric-negative* to find if there is a connection between the numbers reported by our prototype and the verdict in the filled in rubric. For some patterns we record *Apollo-not-found* if the pattern was not matched by our prototype, but the structure is in fact present; *God-class* if the project contains a God class that replaces the main tab, which is relevant for those criteria; and *In-callstack* if event handling variables were used outside the event handling methods, but within methods that were called by those event handlers.

For SS, table B.2, we additionally record *Many-false-positive* if the matches returned by the prototype include a lot of false positives for this criterion.

Table B.1: Agreement between automated and manual assessment of AiC projects

| Program | 2: Randomness | 4: Particles | 5: Flocking | 11: Class interaction | 12: Classes | 14: Global variables | 17: User interaction | 21: Similar code | 23: Unused code |
|----------|---------------|----------------------------|----------------------|--------------------------------|--------------------------|------------------------------------|------------------------|--------------------------|-------------------------|
| 2020/071 | Agree | Disagree, Apollo-not-found | Agree | 12, Rubric-not-mentioned | 12, Rubric-not-mentioned | 7, Rubric-not-mentioned | Agree | 2, Rubric-not-mentioned | 3, Rubric-not-mentioned |
| 2020/072 | Agree | Rubric-not-mentioned | Rubric-not-mentioned | 24, Rubric-negative, God-class | 12, Rubric-not-mentioned | 0, Rubric-not-mentioned, God-class | Agree | 2, Rubric-not-mentioned | 1, Rubric-not-mentioned |
| 2020/076 | Agree | Rubric-not-mentioned | Rubric-not-mentioned | 25, Rubric-positive | 13, Rubric-not-mentioned | 2, Rubric-not-mentioned | Disagree, In-callstack | 2, Rubric-not-mentioned | 0, Rubric-not-mentioned |
| 2020/078 | Agree | Agree | Agree | 3, Rubric-negative | 9, Rubric-not-mentioned | 23, 20-same, Agree | Rubric-not-mentioned | 0, Rubric-not-mentioned | 1, Rubric-not-mentioned |
| 2020/081 | Agree | Agree | Agree | 40, Rubric-negative, God-class | 17, Agree | 0, Rubric-not-mentioned, God-class | Disagree, In-callstack | 28, Rubric-not-mentioned | 4, Rubric-not-mentioned |
| 2020/085 | Disagree | Agree | Agree | 11, Rubric-negative | 12, Rubric-not-mentioned | 3, Rubric-not-mentioned | Agree | 0, Rubric-not-mentioned | 1, Rubric-not-mentioned |
| 2020/086 | Agree | Disagree, Apollo-not-found | Disagree | 5, Rubric-not-mentioned | 14, Rubric-not-mentioned | 5, Rubric-not-mentioned | Agree | 0, Rubric-not-mentioned | 0, Rubric-not-mentioned |

| Program | 2: Randomness | 4: Particles | 5: Flocking | 11: Class interaction | 12: Classes | 14: Global variables | 17: User interaction | 21: Similar code | 23: Unused code |
|----------|---------------|----------------------------|-------------|--------------------------------|--------------------------|------------------------------------|------------------------|-------------------------|-------------------------|
| 2020/088 | Agree | Disagree, Apollo-not-found | Agree | 25, Rubric-negative, God-class | 12, Rubric-not-mentioned | 1, Rubric-not-mentioned | Disagree, In-callstack | 0, Rubric-not-mentioned | 1, Rubric-not-mentioned |
| 2020/089 | Disagree | Agree | Agree | 25, Rubric-negative, God-class | 12, Rubric-not-mentioned | 0, Rubric-not-mentioned, God-class | Rubric-not-mentioned | 0, Rubric-not-mentioned | 0, Rubric-not-mentioned |
| 2020/090 | Agree | Disagree, Apollo-not-found | Agree | 14, Rubric-negative | 12, Rubric-not-mentioned | 6, Agree | Agree | 0, Rubric-not-mentioned | 0, Rubric-not-mentioned |

Table B.2: Agreement between automated and manual assessment of SS projects

| Program | 50: Constants | 59: Server tests | 61: Access state | 62: Private fields | 63: Public methods | 66: One package | 69: Interfaces | 77: Creational pattern | 78: Custom exceptions |
|---------|----------------------|------------------|-------------------------------|--------------------|--------------------|-----------------|----------------------|-------------------------------|-----------------------|
| 020 | Rubric-not-mentioned | Agree | Agree | Agree | Disagree | Agree | Rubric-not-mentioned | Disagree, Many-false-positive | Agree |
| 021 | Rubric-not-mentioned | Agree | Disagree, Many-false-positive | Disagree | Disagree | Agree | Rubric-not-mentioned | Disagree, Many-false-positive | Agree |

| Program | 50:
Constants | 59: Server
tests | 61: Access
state | 62: Private
fields | 63: Public
methods | 66: One
package | 69:
Interfaces | 77:
Creational
pattern | 78:
Custom
exceptions |
|---------|--------------------------|--------------------------|---------------------|-----------------------|--------------------------|--------------------|--------------------------|--------------------------------------|-----------------------------|
| 022 | Agree | Agree | Agree | Agree | Disagree | Agree | Disagree | Disagree,
Many-false-
positive | Rubric-not-
mentioned |
| 023 | Agree | Agree | Agree | Disagree | Disagree | Agree | Disagree | Disagree | Agree |
| 024 | Rubric-not-
mentioned | Agree | Agree | Agree | Disagree | Agree | Rubric-not-
mentioned | Rubric-not-
mentioned | Agree |
| 025 | Agree | Agree | Agree | Agree | Rubric-not-
mentioned | Agree | Disagree | Disagree,
Many-false-
positive | Agree |
| 026 | Rubric-not-
mentioned | Rubric-not-
mentioned | Agree | Agree | Rubric-not-
mentioned | Agree | Disagree | Disagree,
Many-false-
positive | Agree |
| 027 | Agree | Agree | Agree | Agree | Rubric-not-
mentioned | Agree | Rubric-not-
mentioned | Disagree | Agree |
| 028 | Agree | Agree | Agree | Disagree | Agree | Agree | Agree | Disagree,
Many-false-
positive | Agree |
| 029 | Rubric-not-
mentioned | Agree | Disagree | Disagree | Disagree | Agree | Disagree | Disagree,
Many-false-
positive | Agree |